



## **ef\_vi User Guide**

SF-114063-CD , Issue 10

2019/09/26 12:41:00

Xilinx, Inc



## ef\_vi User Guide

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

Onload is licensed under the GNU General Public License (Version 2, June 1991). See the LICENSE file in the distribution for details. The Onload Extensions Stub Library is Copyright licensed under the BSD 2-Clause License.

You will not disclose to a third party the results of any performance tests carried out using Onload, Cloud Onload or EnterpriseOnload without the prior written consent of Xilinx, Inc.

A list of patents associated with this product is at <http://www.solarflare.com/patent>.

### **AUTOMOTIVE APPLICATIONS DISCLAIMER**

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### **Copyright**

© Copyright 2019 Xilinx, Inc. Xilinx, the Xilinx logo, Solarflare, Onload, TCPDirect, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

SF-114063-CD

Last Revised: September 2019

Issue 10



# Contents

<b>1</b>	<b>ef_vi</b>	<b>1</b>
1.1	Introduction . . . . .	1
<b>2</b>	<b>What's New</b>	<b>3</b>
2.1	IPv6 checksum functions . . . . .	3
2.2	Bug fixes . . . . .	3
<b>3</b>	<b>Overview</b>	<b>5</b>
3.1	Capabilities . . . . .	5
3.2	Flexibility . . . . .	6
3.3	Scalability . . . . .	6
3.4	Use cases . . . . .	6
3.4.1	Sockets acceleration . . . . .	6
3.4.2	Packet capture . . . . .	6
3.4.3	Packet replay . . . . .	7
3.4.4	Application as an end-station . . . . .	7
3.4.5	Software defined bridging, switching and routing . . . . .	7
3.5	Activation requirements . . . . .	7
3.5.1	Using sniffing . . . . .	7

<b>4</b>	<b>Concepts</b>	<b>9</b>
4.1	Virtual Interface	9
4.1.1	Virtual Interface Set	10
4.1.2	Event queue	10
4.1.3	Transmit descriptor ring	10
4.1.4	Receive descriptor ring	10
4.2	Protection Domain	11
4.3	Memory Region	11
4.4	Packet Buffer	11
4.4.1	Jumbo Packets	12
4.4.2	Packet Buffer Descriptor	12
4.5	Programmed I/O	12
4.6	Cut-through PIO	12
4.6.1	Underrun, poisoning and fallback	13
4.6.2	CTPIO diagnostics	13
4.7	Filters	13
4.7.1	Multiple Filters	14
4.8	Virtual LANs	14
4.9	TX Alternatives	14
<b>5</b>	<b>Example Applications</b>	<b>15</b>
5.1	eflatency	15
5.1.1	Usage	16
5.2	efsend	16
5.3	efsink	16
5.3.1	Usage	17
5.4	efforward	17
5.5	efsend_timestamping	17
5.6	efsend_pio	17
5.7	efsend_pio_warm	18
5.8	efsink_packed	18
5.9	efforward_packed	18
5.10	efrss	18
5.11	efdelegated_client	18
5.12	efjumborx	19
5.13	exchange	19
5.14	trader_onload_ds_efvi	19
5.14.1	Usage	19
5.15	efrink_controller	20
5.16	efrink_consumer	20
5.17	Building the Example Applications	20

<b>6</b>	<b>Using ef_vi</b>	<b>21</b>
6.1	Components	21
6.2	Compiling and Linking	21
6.3	Setup	22
6.3.1	Using Virtual Interface Sets	23
6.4	Creating packet buffers	23
6.4.1	Buffer Tables	23
6.5	Transmitting Packets	24
6.5.1	Transmitting Jumbo Frames	25
6.5.2	Programmed I/O	25
6.5.3	Cut-through PIO	25
6.5.4	TX Alternatives	26
6.6	Handling Events	27
6.6.1	Blocking on a file descriptor	27
6.7	Receiving packets	28
6.7.1	Finding the Packet Data	28
6.7.2	Receiving Jumbo Packets	30
6.7.3	RX Event Merging	31
6.8	Adding Filters	35
6.8.1	Filter permission requirements	36
6.8.2	Filters available per Firmware Variant	37
6.9	IGMP subscriptions	38
6.10	Freeing Resources	39
6.11	Design Considerations	40
6.11.1	Interrupts	40
6.11.2	Thread Safety	40
6.11.3	Packet Buffer Addressing	40
6.11.4	Virtual machines	41
6.12	Known Limitations	41
6.12.1	Timestamping	41
6.12.2	Minimum Fill Level	41
6.13	Example	42

<b>7</b>	<b>Worked Example</b>	<b>43</b>
7.1	Setup	43
7.2	Creating Packet buffers	44
7.3	Adding Filters	44
7.4	Receiving packets	45
7.5	Handling Events	45
7.6	Transmitting packets	46
<b>8</b>	<b>Data Structure Index</b>	<b>47</b>
8.1	Data Structures	47
<b>9</b>	<b>File Index</b>	<b>49</b>
9.1	File List	49
<b>10</b>	<b>Data Structure Documentation</b>	<b>51</b>
10.1	ef_event Union Reference	51
10.1.1	Detailed Description	53
10.1.2	Field Documentation	53
10.1.2.1	generic	53
10.1.2.2	rx	53
10.1.2.3	rx_discard	53
10.1.2.4	rx_multi	53
10.1.2.5	rx_multi_discard	54
10.1.2.6	rx_no_desc_trunc	54
10.1.2.7	rx_packed_stream	54
10.1.2.8	sw	54
10.1.2.9	tx	54
10.1.2.10	tx_alt	54
10.1.2.11	tx_error	54
10.1.2.12	tx_timestamp	55
10.2	ef_eventq_state Struct Reference	55



10.2.1	Detailed Description	55
10.2.2	Field Documentation	55
10.2.2.1	evq_clear_stride	55
10.2.2.2	evq_ptr	56
10.2.2.3	sync_flags	56
10.2.2.4	sync_timestamp_major	56
10.2.2.5	sync_timestamp_minimum	56
10.2.2.6	sync_timestamp_minor	56
10.2.2.7	sync_timestamp_synchronised	57
10.3	ef_filter_cookie Struct Reference	57
10.3.1	Detailed Description	57
10.3.2	Field Documentation	57
10.3.2.1	filter_id	57
10.3.2.2	filter_type	58
10.4	ef_filter_spec Struct Reference	58
10.4.1	Detailed Description	58
10.4.2	Field Documentation	58
10.4.2.1	data	58
10.4.2.2	flags	59
10.4.2.3	type	59
10.5	ef_iovec Struct Reference	59
10.5.1	Detailed Description	59
10.5.2	Member Function Documentation	60
10.5.2.1	EF_VI_ALIGN()	60
10.5.3	Field Documentation	60
10.5.3.1	iov_len	60
10.6	ef_memreg Struct Reference	60
10.6.1	Detailed Description	60
10.6.2	Field Documentation	61

10.6.2.1	mr_dma_addrs	61
10.6.2.2	mr_dma_addrs_base	61
10.7	ef_packed_stream_packet Struct Reference	61
10.7.1	Detailed Description	61
10.7.2	Field Documentation	62
10.7.2.1	ps_cap_len	62
10.7.2.2	ps_flags	62
10.7.2.3	ps_next_offset	62
10.7.2.4	ps_orig_len	62
10.7.2.5	ps_pkt_start_offset	63
10.7.2.6	ps_ts_nsec	63
10.7.2.7	ps_ts_sec	63
10.8	ef_packed_stream_params Struct Reference	63
10.8.1	Detailed Description	64
10.8.2	Field Documentation	64
10.8.2.1	psp_buffer_align	64
10.8.2.2	psp_buffer_size	64
10.8.2.3	psp_max_usable_buffers	64
10.8.2.4	psp_start_offset	65
10.9	ef_pd Struct Reference	65
10.9.1	Detailed Description	65
10.9.2	Field Documentation	65
10.9.2.1	pd_cluster_dh	65
10.9.2.2	pd_cluster_name	66
10.9.2.3	pd_cluster_sock	66
10.9.2.4	pd_cluster_viset_index	66
10.9.2.5	pd_cluster_viset_resource_id	66
10.9.2.6	pd_flags	66
10.9.2.7	pd_intf_name	67

10.9.2.8 pd_resource_id . . . . .	67
10.10ef_pio Struct Reference . . . . .	67
10.10.1 Detailed Description . . . . .	67
10.10.2 Field Documentation . . . . .	67
10.10.2.1 pio_buffer . . . . .	68
10.10.2.2 pio_io . . . . .	68
10.10.2.3 pio_len . . . . .	68
10.10.2.4 pio_resource_id . . . . .	68
10.11ef_vi Struct Reference . . . . .	68
10.11.1 Detailed Description . . . . .	70
10.11.2 Field Documentation . . . . .	70
10.11.2.1 ep_state . . . . .	70
10.11.2.2 evq_base . . . . .	70
10.11.2.3 evq_mask . . . . .	70
10.11.2.4 inited . . . . .	71
10.11.2.5 io . . . . .	71
10.11.2.6 linked_pio . . . . .	71
10.11.2.7 nic_type . . . . .	71
10.11.2.8 ops . . . . .	71
10.11.2.9 rx_buffer_len . . . . .	72
10.11.2.10rx_discard_mask . . . . .	72
10.11.2.11rx_prefix_len . . . . .	72
10.11.2.12rx_ts_correction . . . . .	72
10.11.2.13timer_quantum_ns . . . . .	72
10.11.2.14ts_format . . . . .	73
10.11.2.15tx_alt_hw2id . . . . .	73
10.11.2.16tx_alt_id2hw . . . . .	73
10.11.2.17tx_alt_num . . . . .	73
10.11.2.18tx_push_thresh . . . . .	73

10.11.2.19	x_ts_correction_ns	74
10.11.2.20	vi_clustered	74
10.11.2.21	vi_ctpio_mmap_ptr	74
10.11.2.22	vi_ctpio_wb_ticks	74
10.11.2.23	vi_flags	74
10.11.2.24	vi_i	75
10.11.2.25	vi_io_mmap_bytes	75
10.11.2.26	vi_io_mmap_ptr	75
10.11.2.27	vi_is_normal	75
10.11.2.28	vi_is_packed_stream	75
10.11.2.29	vi_mem_mmap_bytes	76
10.11.2.30	vi_mem_mmap_ptr	76
10.11.2.31	vi_out_flags	76
10.11.2.32	vi_ps_buf_size	76
10.11.2.33	vi_qs	76
10.11.2.34	vi_qs_n	77
10.11.2.35	vi_resource_id	77
10.11.2.36	vi_rxq	77
10.11.2.37	vi_stats	77
10.11.2.38	vi_txq	77
10.12	ef_vi_layout_entry Struct Reference	78
10.12.1	Detailed Description	78
10.12.2	Field Documentation	78
10.12.2.1	evle_description	78
10.12.2.2	evle_offset	78
10.12.2.3	evle_type	79
10.13	ef_vi_nic_type Struct Reference	79
10.13.1	Detailed Description	79
10.13.2	Field Documentation	79

10.13.2.1 arch . . . . .	79
10.13.2.2 nic_flags . . . . .	80
10.13.2.3 revision . . . . .	80
10.13.2.4 variant . . . . .	80
10.14ef_vi_rxq Struct Reference . . . . .	80
10.14.1 Detailed Description . . . . .	81
10.14.2 Field Documentation . . . . .	81
10.14.2.1 descriptors . . . . .	81
10.14.2.2 ids . . . . .	81
10.14.2.3 mask . . . . .	81
10.15ef_vi_rxq_state Struct Reference . . . . .	82
10.15.1 Detailed Description . . . . .	82
10.15.2 Field Documentation . . . . .	82
10.15.2.1 added . . . . .	82
10.15.2.2 bytes_acc . . . . .	82
10.15.2.3 in_jumbo . . . . .	83
10.15.2.4 last_desc_i . . . . .	83
10.15.2.5 posted . . . . .	83
10.15.2.6 removed . . . . .	83
10.15.2.7 rx_ps_credit_avail . . . . .	83
10.16ef_vi_set Struct Reference . . . . .	84
10.16.1 Detailed Description . . . . .	84
10.16.2 Field Documentation . . . . .	84
10.16.2.1 vis_pd . . . . .	84
10.16.2.2 vis_res_id . . . . .	84
10.17ef_vi_state Struct Reference . . . . .	85
10.17.1 Detailed Description . . . . .	85
10.17.2 Field Documentation . . . . .	85
10.17.2.1 evq . . . . .	85

10.17.2.2 rxq . . . . .	85
10.17.2.3 txq . . . . .	86
10.18ef_vi_stats Struct Reference . . . . .	86
10.18.1 Detailed Description . . . . .	86
10.18.2 Field Documentation . . . . .	86
10.18.2.1 evq_gap . . . . .	86
10.18.2.2 rx_ev_bad_desc_i . . . . .	87
10.18.2.3 rx_ev_bad_q_label . . . . .	87
10.18.2.4 rx_ev_lost . . . . .	87
10.19ef_vi_stats_field_layout Struct Reference . . . . .	87
10.19.1 Detailed Description . . . . .	88
10.19.2 Field Documentation . . . . .	88
10.19.2.1 evsfl_name . . . . .	88
10.19.2.2 evsfl_offset . . . . .	88
10.19.2.3 evsfl_size . . . . .	88
10.20ef_vi_stats_layout Struct Reference . . . . .	88
10.20.1 Detailed Description . . . . .	89
10.20.2 Field Documentation . . . . .	89
10.20.2.1 evsl_data_size . . . . .	89
10.20.2.2 evsl_fields . . . . .	89
10.20.2.3 evsl_fields_num . . . . .	89
10.21ef_vi_transmit_alt_overhead Struct Reference . . . . .	90
10.21.1 Detailed Description . . . . .	90
10.21.2 Field Documentation . . . . .	90
10.21.2.1 mask . . . . .	90
10.21.2.2 post_round . . . . .	90
10.21.2.3 pre_round . . . . .	91
10.22ef_vi_txq Struct Reference . . . . .	91
10.22.1 Detailed Description . . . . .	91

10.22.2 Field Documentation . . . . .	91
10.22.2.1 descriptors . . . . .	91
10.22.2.2 ids . . . . .	92
10.22.2.3 mask . . . . .	92
10.23 ef_vi_txq_state Struct Reference . . . . .	92
10.23.1 Detailed Description . . . . .	92
10.23.2 Field Documentation . . . . .	92
10.23.2.1 added . . . . .	93
10.23.2.2 previous . . . . .	93
10.23.2.3 removed . . . . .	93
10.23.2.4 ts_nsec . . . . .	93
10.24 ef_vi::ops Struct Reference . . . . .	93
10.24.1 Detailed Description . . . . .	94
10.24.2 Field Documentation . . . . .	94
10.24.2.1 eventq_poll . . . . .	94
10.24.2.2 eventq_prime . . . . .	95
10.24.2.3 eventq_timer_clear . . . . .	95
10.24.2.4 eventq_timer_prime . . . . .	95
10.24.2.5 eventq_timer_run . . . . .	95
10.24.2.6 eventq_timer_zero . . . . .	95
10.24.2.7 receive_init . . . . .	96
10.24.2.8 receive_push . . . . .	96
10.24.2.9 transmit . . . . .	96
10.24.2.10 transmit_alt_discard . . . . .	96
10.24.2.11 transmit_alt_go . . . . .	96
10.24.2.12 transmit_alt_select . . . . .	97
10.24.2.13 transmit_alt_select_default . . . . .	97
10.24.2.14 transmit_alt_stop . . . . .	97
10.24.2.15 transmit_copy_pio . . . . .	97
10.24.2.16 transmit_copy_pio_warm . . . . .	97
10.24.2.17 transmit_pio . . . . .	98
10.24.2.18 transmit_pio_warm . . . . .	98
10.24.2.19 transmit_push . . . . .	98
10.24.2.20 transmitv . . . . .	98
10.24.2.21 transmitv_ctpio . . . . .	98
10.24.2.22 transmitv_ctpio_copy . . . . .	99
10.24.2.23 transmitv_init . . . . .	99

<b>11 File Documentation</b>	<b>101</b>
11.1 000_main.dox File Reference	101
11.1.1 Detailed Description	101
11.2 005_whats_new.dox File Reference	101
11.2.1 Detailed Description	102
11.3 010_overview.dox File Reference	102
11.3.1 Detailed Description	102
11.4 020_concepts.dox File Reference	102
11.4.1 Detailed Description	103
11.5 030_apps.dox File Reference	103
11.5.1 Detailed Description	103
11.6 040_using.dox File Reference	103
11.6.1 Detailed Description	104
11.7 050_examples.dox File Reference	104
11.7.1 Detailed Description	104
11.8 base.h File Reference	104
11.8.1 Detailed Description	105
11.8.2 Function Documentation	105
11.8.2.1 ef_driver_close()	105
11.8.2.2 ef_driver_open()	106
11.8.2.3 ef_eventq_wait()	106
11.9 capabilities.h File Reference	107
11.9.1 Detailed Description	108
11.9.2 Enumeration Type Documentation	108
11.9.2.1 ef_vi_capability	108
11.9.3 Function Documentation	110
11.9.3.1 ef_vi_capabilities_get()	110
11.9.3.2 ef_vi_capabilities_max()	110
11.9.3.3 ef_vi_capabilities_name()	110



11.10checksum.h File Reference . . . . .	111
11.10.1 Detailed Description . . . . .	112
11.10.2 Function Documentation . . . . .	112
11.10.2.1 ef_icmpv6_checksum() . . . . .	112
11.10.2.2 ef_ip_checksum() . . . . .	112
11.10.2.3 ef_tcp_checksum() . . . . .	113
11.10.2.4 ef_tcp_checksum_ip6() . . . . .	113
11.10.2.5 ef_tcp_checksum_ipx() . . . . .	114
11.10.2.6 ef_udp_checksum() . . . . .	114
11.10.2.7 ef_udp_checksum_ip6() . . . . .	114
11.10.2.8 ef_udp_checksum_ipx() . . . . .	115
11.11ef_vi.h File Reference . . . . .	115
11.11.1 Detailed Description . . . . .	122
11.11.2 Macro Definition Documentation . . . . .	122
11.11.2.1 EF_EVENT_RX_MULTI_CONT . . . . .	123
11.11.2.2 EF_EVENT_RX_MULTI_SOP . . . . .	123
11.11.2.3 EF_EVENT_RX_PS_NEXT_BUFFER . . . . .	123
11.11.2.4 ef_eventq_poll . . . . .	123
11.11.2.5 ef_eventq_prime . . . . .	124
11.11.2.6 EF_VI_CTPIO_CT_THRESHOLD_SNF . . . . .	124
11.11.2.7 ef_vi_receive_init . . . . .	124
11.11.2.8 ef_vi_receive_push . . . . .	125
11.11.2.9 ef_vi_transmit . . . . .	125
11.11.2.10ef_vi_transmit_alt_discard . . . . .	126
11.11.2.11ef_vi_transmit_alt_go . . . . .	126
11.11.2.12ef_vi_transmit_alt_select . . . . .	127
11.11.2.13ef_vi_transmit_alt_select_normal . . . . .	127
11.11.2.14ef_vi_transmit_alt_stop . . . . .	128
11.11.2.15ef_vi_transmit_copy_pio . . . . .	128

11.11.2.16	ef_vi_transmit_copy_pio_warm . . . . .	129
11.11.2.17	ef_vi_transmit_pio . . . . .	130
11.11.2.18	ef_vi_transmit_pio_warm . . . . .	131
11.11.2.19	ef_vi_transmit_push . . . . .	131
11.11.2.20	ef_vi_transmitv . . . . .	132
11.11.2.21	ef_vi_transmitv_ctpio . . . . .	132
11.11.2.22	ef_vi_transmitv_ctpio_copy . . . . .	133
11.11.2.23	ef_vi_transmitv_init . . . . .	134
11.11.3	Typedef Documentation . . . . .	135
11.11.3.1	ef_request_id . . . . .	135
11.11.3.2	ef_vi . . . . .	135
11.11.4	Enumeration Type Documentation . . . . .	135
11.11.4.1	anonymous enum . . . . .	135
11.11.4.2	anonymous enum . . . . .	136
11.11.4.3	anonymous enum . . . . .	136
11.11.4.4	ef_vi_arch . . . . .	136
11.11.4.5	ef_vi_flags . . . . .	137
11.11.4.6	ef_vi_layout_type . . . . .	138
11.11.4.7	ef_vi_out_flags . . . . .	138
11.11.4.8	ef_vi_rx_discard_err_flags . . . . .	138
11.11.5	Function Documentation . . . . .	139
11.11.5.1	ef_eventq_capacity() . . . . .	139
11.11.5.2	ef_eventq_current() . . . . .	139
11.11.5.3	ef_eventq_has_event() . . . . .	140
11.11.5.4	ef_eventq_has_many_events() . . . . .	140
11.11.5.5	ef_vi_driver_interface_str() . . . . .	141
11.11.5.6	ef_vi_flags() . . . . .	141
11.11.5.7	ef_vi_instance() . . . . .	141
11.11.5.8	ef_vi_receive_buffer_len() . . . . .	141

11.11.5.9 ef_vi_receive_capacity()	142
11.11.5.10 ef_vi_receive_fill_level()	142
11.11.5.11 ef_vi_receive_get_bytes()	143
11.11.5.12 ef_vi_receive_get_timestamp()	143
11.11.5.13 ef_vi_receive_get_timestamp_with_sync_flags()	144
11.11.5.14 ef_vi_receive_post()	145
11.11.5.15 ef_vi_receive_prefix_len()	146
11.11.5.16 ef_vi_receive_query_layout()	146
11.11.5.17 ef_vi_receive_set_buffer_len()	146
11.11.5.18 ef_vi_receive_set_discards()	147
11.11.5.19 ef_vi_receive_space()	147
11.11.5.20 ef_vi_receive_unbundle()	148
11.11.5.21 ef_vi_resource_id()	148
11.11.5.22 ef_vi_set_tx_push_threshold()	148
11.11.5.23 ef_vi_transmit_alt_num_ids()	149
11.11.5.24 ef_vi_transmit_alt_query_overhead()	149
11.11.5.25 ef_vi_transmit_alt_usage()	150
11.11.5.26 ef_vi_transmit_capacity()	150
11.11.5.27 ef_vi_transmit_ctpio()	151
11.11.5.28 ef_vi_transmit_ctpio_fallback()	151
11.11.5.29 ef_vi_transmit_fill_level()	152
11.11.5.30 ef_vi_transmit_init()	152
11.11.5.31 ef_vi_transmit_init_undo()	153
11.11.5.32 ef_vi_transmit_space()	153
11.11.5.33 ef_vi_transmit_unbundle()	153
11.11.5.34 ef_vi_transmitv_ctpio_fallback()	154
11.11.5.35 ef_vi_version_str()	155
11.12 memreg.h File Reference	155
11.12.1 Detailed Description	156

11.12.2 Function Documentation . . . . .	156
11.12.2.1 ef_memreg_alloc() . . . . .	156
11.12.2.2 ef_memreg_dma_addr() . . . . .	157
11.12.2.3 ef_memreg_free() . . . . .	157
11.13 packedstream.h File Reference . . . . .	158
11.13.1 Detailed Description . . . . .	159
11.13.2 Macro Definition Documentation . . . . .	159
11.13.2.1 EF_VI_PS_FLAG_BAD_IP_CSUM . . . . .	159
11.13.3 Function Documentation . . . . .	160
11.13.3.1 ef_packed_stream_packet_first() . . . . .	160
11.13.3.2 ef_packed_stream_packet_next() . . . . .	160
11.13.3.3 ef_packed_stream_packet_payload() . . . . .	160
11.13.3.4 ef_vi_packed_stream_get_params() . . . . .	161
11.13.3.5 ef_vi_packed_stream_unbundle() . . . . .	161
11.14 pd.h File Reference . . . . .	162
11.14.1 Detailed Description . . . . .	163
11.14.2 Enumeration Type Documentation . . . . .	163
11.14.2.1 ef_pd_flags . . . . .	163
11.14.3 Function Documentation . . . . .	164
11.14.3.1 ef_pd_alloc() . . . . .	164
11.14.3.2 ef_pd_alloc_by_name() . . . . .	164
11.14.3.3 ef_pd_alloc_with_vport() . . . . .	165
11.14.3.4 ef_pd_free() . . . . .	165
11.14.3.5 ef_pd_interface_name() . . . . .	166
11.15 pio.h File Reference . . . . .	166
11.15.1 Detailed Description . . . . .	167
11.15.2 Function Documentation . . . . .	167
11.15.2.1 ef_pio_alloc() . . . . .	168
11.15.2.2 ef_pio_free() . . . . .	169

11.15.2.3 ef_pio_link_vi()	169
11.15.2.4 ef_pio_memcpy()	170
11.15.2.5 ef_pio_unlink_vi()	171
11.15.2.6 ef_vi_get_pio_size()	171
11.16timer.h File Reference	171
11.16.1 Detailed Description	172
11.16.2 Macro Definition Documentation	172
11.16.2.1 ef_eventq_timer_clear	172
11.16.2.2 ef_eventq_timer_prime	173
11.16.2.3 ef_eventq_timer_run	173
11.16.2.4 ef_eventq_timer_zero	174
11.17vi.h File Reference	175
11.17.1 Detailed Description	177
11.17.2 Enumeration Type Documentation	177
11.17.2.1 anonymous enum	177
11.17.2.2 ef_filter_flags	178
11.17.3 Function Documentation	178
11.17.3.1 ef_eventq_put()	178
11.17.3.2 ef_filter_spec_init()	179
11.17.3.3 ef_filter_spec_set_block_kernel()	180
11.17.3.4 ef_filter_spec_set_block_kernel_multicast()	180
11.17.3.5 ef_filter_spec_set_block_kernel_unicast()	181
11.17.3.6 ef_filter_spec_set_eth_local()	181
11.17.3.7 ef_filter_spec_set_eth_type()	182
11.17.3.8 ef_filter_spec_set_ip4_full()	182
11.17.3.9 ef_filter_spec_set_ip4_local()	183
11.17.3.10ef_filter_spec_set_ip6_full()	183
11.17.3.11ef_filter_spec_set_ip6_local()	184
11.17.3.12ef_filter_spec_set_ip_proto()	185

11.17.3.13ef_filter_spec_set_multicast_all()	185
11.17.3.14ef_filter_spec_set_multicast_mismatch()	185
11.17.3.15ef_filter_spec_set_port_sniff()	186
11.17.3.16ef_filter_spec_set_tx_port_sniff()	186
11.17.3.17ef_filter_spec_set_unicast_all()	187
11.17.3.18ef_filter_spec_set_unicast_mismatch()	187
11.17.3.19ef_filter_spec_set_vlan()	188
11.17.3.20ef_vi_alloc_from_pd()	188
11.17.3.21ef_vi_alloc_from_set()	190
11.17.3.22ef_vi_filter_add()	191
11.17.3.23ef_vi_filter_del()	191
11.17.3.24ef_vi_flush()	192
11.17.3.25ef_vi_free()	192
11.17.3.26ef_vi_get_mac()	193
11.17.3.27ef_vi_mtu()	193
11.17.3.28ef_vi_pace()	194
11.17.3.29ef_vi_prime()	194
11.17.3.30ef_vi_set_alloc_from_pd()	195
11.17.3.31ef_vi_set_filter_add()	195
11.17.3.32ef_vi_set_filter_del()	196
11.17.3.33ef_vi_set_free()	196
11.17.3.34ef_vi_stats_query()	197
11.17.3.35ef_vi_stats_query_layout()	197
11.17.3.36ef_vi_transmit_alt_alloc()	198
11.17.3.37ef_vi_transmit_alt_free()	198
11.17.3.38ef_vi_transmit_alt_query_buffering()	199

# Chapter 1

## ef\_vi

Solarflare's ef\_vi API is a flexible interface for passing Ethernet frames between applications and the network. It is the internal API used by Onload for sending and receiving packets.

### 1.1 Introduction

ef\_vi grants an application direct access to the Solarflare network adapter datapath to deliver lower latency and reduced per message processing overheads. It can be used directly by applications that want the very lowest latency send and receive API, and that do not require a POSIX socket interface.

The key features of ef\_vi are:

- **User-space:** Ef\_vi can be used by unprivileged user-space applications.
- **Kernel bypass:** Data path operations do not require system calls.
- **Low CPU overhead:** Data path operations consume very few CPU cycles.
- **Low latency:** Suitable for ultra-low latency applications.
- **High packet rates:** Supports millions of packets per second per core.
- **Zero-copy:** Particularly efficient for filtering and forwarding applications.
- **Flexibility:** Supports many use cases (see [Use cases](#)).
- **Redistributable:** ef\_vi is free software distributed under a LGPL license.

Each ef\_vi instance provides a [Virtual Interface](#) for an application to use:

- Each virtual interface provides a TX channel for passing packets to the adapter. Packets may be transmitted onto the wire, or looped-back in the adapter for delivery back into the host, or both.
- Each virtual interface also provides an RX channel for receiving packets from the adapter. These can be packets received from the wire, or looped-back from the TX path.





## Chapter 2

# What's New

This chapter tells you what's new in this release of [ef\\_vi](#).

If you are not a previous user of [ef\\_vi](#), go to [Overview](#).

### 2.1 IPv6 checksum functions

New functions have been added for calculating IPv6 checksums.

### 2.2 Bug fixes

Various bugs have been fixed. For details, see the [ChangeLog](#) file.



## Chapter 3

# Overview

This part of the documentation gives an overview of ef\_vi and how it is often used.

### 3.1 Capabilities

Ef\_vi is a low level OSI level 2 interface which sends and receives raw Ethernet frames. It is essentially a thin wrapper around the VNIC (virtual network interface controller) interface offered by the network adapters. It exposes, and can be used with, many of the advanced capabilities of Solarflare network adapters, including:

- Checksum offloads
- Hardware time stamping (RX and TX)
- Switching functions, including:
  - multicast replication
  - loopback
  - VLAN tag insert/strip
- Load spreading by hashing (also known as receive-side scaling) and flow steering
- Data path sniffing
- PIO mode for low latency
- CTPIO mode for ultra-low latency
- Scatter gather transmit and receive
- PCI pass-through and SR-IOV for virtualized environments.

But because the ef\_vi API operates at this low level, any application using it must implement the higher layer protocols itself, and also deal with any exceptions or other unusual conditions.

## 3.2 Flexibility

A key advantage of ef\_vi when compared to other similar technologies is the flexibility it offers. Each ef\_vi instance can be thought of as a virtual port on the network adapter's switch. Ef\_vi can be used to handle all packets associated with a physical port, or just a subset. This means that ef\_vi can be used in parallel with the standard Linux kernel network stack and other acceleration technologies such as Solarflare's OpenOnload sockets acceleration. For example, a trading application might use ef\_vi to receive UDP market data, but use Onload sockets to make TCP trades.

## 3.3 Scalability

Ef\_vi is also very scalable. Each physical network port on Solarflare's network adapters supports up to 1024 VNIC interfaces. There can be many independent channels between software and the adapter, which can be used to spread load over many cores, or to accelerate large numbers of virtual machines and applications simultaneously.

## 3.4 Use cases

This section gives some examples of how ef\_vi can and is being used:

### 3.4.1 Sockets acceleration

Ef\_vi can be used to replace the BSD sockets API, or another API, for sending and receiving streams of traffic. A common example is handling multicast UDP datagrams in electronic trading systems, where low latency is needed and message rates can be very high.

In this scenario the application establishes an ef\_vi instance, and specifies which packets it would like to receive via this path. For example, an application can select UDP packets with a given destination IP address and port number. All other packets arriving at the network interface continue to be delivered to the regular driver in the kernel stack via a separate path, so only the packets that need to be accelerated are handled by ef\_vi.

Applications can create multiple ef\_vi instances if needed to handle different streams of packets, or to spread load over multiple threads. If transmitting threads each have their own ef\_vi instance then they can transmit packets concurrently without interlocking and without sharing state. This improves efficiency considerably.

### 3.4.2 Packet capture

Solarflare's SolarCapture software is built on top of the ef\_vi API. Like traditional capture APIs ef\_vi can be used to capture all of the packets arriving at a network port, or a subset. Ef\_vi can capture traffic from a 10 gigabit link at line rate at all packet sizes with a single core, and can provide a hardware timestamp for each captured packet.

In "sniffing" mode an ef\_vi instance receives a copy of packets transmitted and/or received by other applications on the host.

### 3.4.3 Packet replay

Ef\_vi can transmit packets from host memory to the network at very high rates, so can be used to construct high performance packet replay applications. (This is another feature of the SolarCapture software).

### 3.4.4 Application as an end-station

Ef\_vi can select packets by destination MAC address, which allows an application to behave as if it were a separate end-station on the Ethernet network. This can be used to implement arbitrary protocols over Ethernet, or to develop applications that simulate behaviour of an end station for benchmarking or test purposes.

Other applications and virtual machines on the host can use the adapter at the same time, via kernel stack, via ef\_vi or via OpenOnload. The application simulating an end-station can communicate with those applications via the adapter, as well as with other remote applications over the physical network.

### 3.4.5 Software defined bridging, switching and routing

Ef\_vi is ideally suited to applications that forward packets between physical or virtual network ports. Zero-copy makes the forwarding path very efficient, and forwarding can be achieved with very low latency.

An ef\_vi instance can be configured to receive "mismatching" packets. That is, all packets not wanted by other applications or virtual machines on the same host. This makes it possible to forward packets to another network segment without knowing the MAC addresses involved, and without cutting the host OS off from the network.

Applications include: High performance firewall applications, network monitoring, intrusion detection, and custom switching and routing. Packets can be forwarded between physical ports and/or between virtual ports. (Virtual ports are logical network ports associated with virtual machines using PCI pass-through).

Ef\_vi is particularly well suited to Network Functions Virtualization (NFV).

## 3.5 Activation requirements

The activation requirements for ef\_vi depend on the network adapter being used:

- If an SFN7000 series adapter is being used, it must have an Onload activation key installed.
- If an SFN8000 or X2 series adapter is being used, and the fastest performance is required, an Onload activation key must be installed.

Without an Onload activation key, the EF\_VI\_RX\_EVENT\_MERGE flag must be set when allocating a virtual interface. This typically reduces performance.

### 3.5.1 Using sniffing

A SolarCapture Pro activation key is always required to use "sniffing" mode, whatever network adapter is used.



# Chapter 4

## Concepts

This part of the documentation describes the concepts involved in ef\_vi.

### 4.1 Virtual Interface

Each ef\_vi instance provides a *virtual interface* to the network adapter.

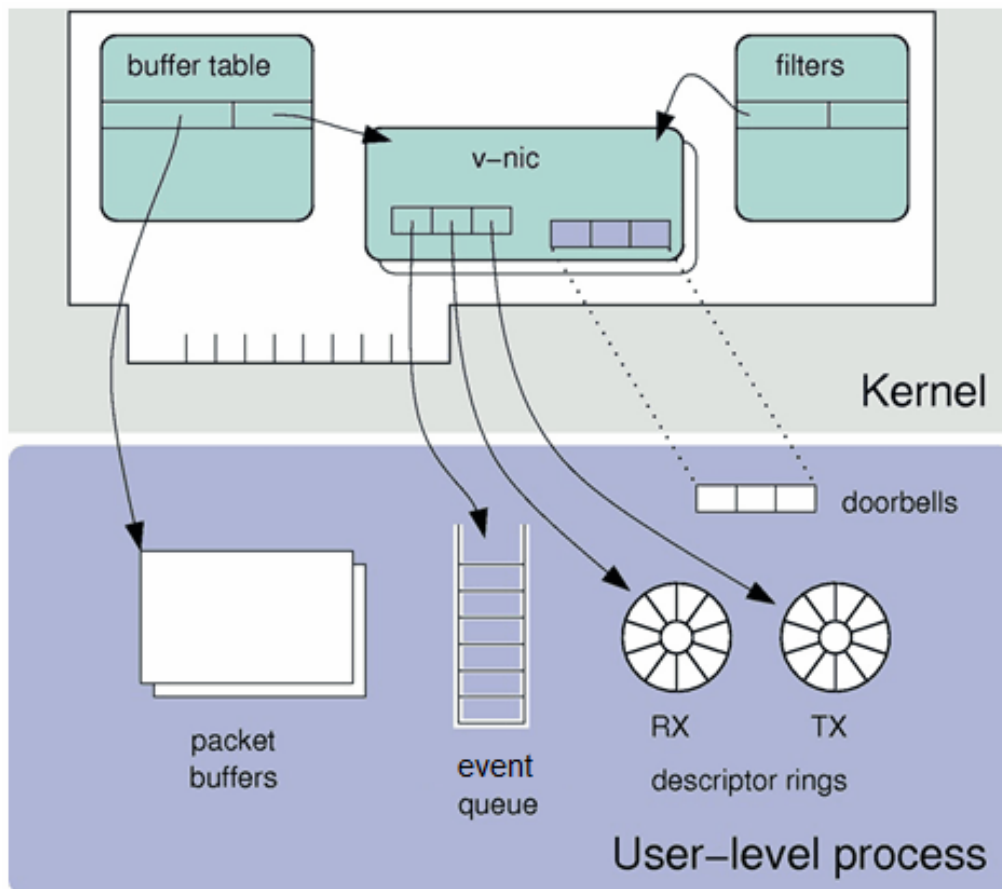


Figure 4.1 Virtual Interface Components

A virtual interface includes the following components:

- an [Event queue](#)
- a [Transmit descriptor ring](#)
- a [Receive descriptor ring](#).

A virtual interface can be allocated one of each of these components, but if the event queue is omitted an alternative virtual interface that has an event queue must be specified.

A virtual interface also has a few hardware resources:

- a doorbell register to inform the card that new RX buffers are available for it to use
- a doorbell register to inform the card that new TX buffers are ready for it to send
- some timers
- a share of an interrupt.

#### 4.1.1 Virtual Interface Set.

A set of virtual interfaces can be created, to distribute load on the matching filters automatically, via *receive side scaling* (RSS).

##### Note

For this to be of use, multiple filters or a wildcard filter are required. A single stream filter will have the same RSS hash for all packets. See [Filters](#).

#### 4.1.2 Event queue

An *event queue* is a channel for passing information from the network adapter to the application. It is used to notify the application of events such as the arrival of packets.

#### 4.1.3 Transmit descriptor ring

The *transmit descriptor ring* is used to pass packets from the application to the adapter. Each entry in the ring is a descriptor which references a buffer containing packet data. Each packet is described by one or more descriptors.

The transmission of packets proceeds in the background, and the adapter notifies the application when they have finished via the event queue.

#### 4.1.4 Receive descriptor ring

The *receive descriptor ring* is used to pass packets from the adapter to the application. The application must pre-allocate buffers and post them to the receive descriptor ring. Each entry in the ring is a descriptor which references a 'free' buffer that the adapter can place a packet into.

When the adapter delivers a packet to an ef\_vi instance, it copies the packet data into the next available receive buffer and notifies the application via the event queue. Large packets can be scattered over multiple receive buffers.



## 4.2 Protection Domain

A *protection domain* identifies a separate address space for the DMA addresses passed to the adapter. It is used to protect multiple ef\_vi applications from one another, or to allow them to share resources:

- Each [Virtual Interface](#) is associated with one protection domain.
- Multiple VIs can be associated with the same protection domain.
- Each [Memory Region](#) is registered with one or more protection domains.
- A memory region can only be used by a virtual interface that is in the same protection domain.
- Memory regions that are registered with multiple protection domains can be used as a shared resource, for example for zero-copy forwarding. See also [Packet Buffer Addressing](#).

### Note

Traditionally device drivers pass the physical addresses of memory buffers to I/O devices. This is usually acceptable because device drivers run within the privileged kernel, and so are isolated from untrusted user-space applications.

Applications using ef\_vi cannot in general use unprotected physical addresses, because by manipulating those addresses prior to passing them to the adapter it would be possible to access memory and devices not otherwise accessible by the application. Protection domains are used to solve this problem.

## 4.3 Memory Region

Any *memory region* used for transmit or receive buffers must be *registered* using the [ef\\_memreg](#) interface. This ensures the memory region meets the requirements of ef\_vi:

- The memory is pinned, so that it can't be swapped out to disk.
- The memory is mapped for DMA, so that the network adapter can access it. The adapter translates the DMA addresses provided by the application to I/O addresses used on the PCIe bus.
- The memory region is page-aligned, for performance.
- The size of the memory region is a multiple of the packet buffer size, so no memory is wasted.

## 4.4 Packet Buffer

A *packet buffer* is a memory allocations on the host which the card will read from when sending packets, or write to when receiving packets. They are usually 2KB in size.

Packets buffers are mapped by the card in such a way that only virtual interfaces in the same protection domain can access them, unless physical addressing mode is explicitly requested. (This feature can only be granted to a group of users by the root user setting an option on the driver.)

#### 4.4.1 Jumbo Packets

Typically, some portion of a packet buffer will be used for meta-data, leaving enough space for a standard sized packet. On receive, large packets will be spread out over multiple packet buffers.

When sending, multiple buffers may be used either to accommodate larger sends, or for convenience (for example: splitting off a standard header that is common to multiple packets).

#### 4.4.2 Packet Buffer Descriptor

Each packet buffer is referred to by a descriptor, which contains:

- a pointer
- an offset
- a length.

It is those descriptors which are actually placed onto the receive and transmit descriptor rings.

### 4.5 Programmed I/O

The `ef_pio` interface exposes a region of memory on the network adapter that can be used for low-latency sends. When using this interface packet data is pushed to the adapter using CPU instructions, instead of being pulled by the adapter using DMA. This reduces latency because it avoids the latency associated with a DMA read.

Applications can get even better latency by writing packet data to the adapter in advance, before the latency critical path. On the critical path the packet data can optionally be updated before being transmitted. This improves latency because it reduces the amount of data that needs to be passed to the adapter on the critical path.

### 4.6 Cut-through PIO

CTPIO (Cut-through PIO) improves send latency by moving packets from the PCIe bus to network port with minimal latency. It can be used in three modes:

1. Cut-through: The frame is transmitted onto the network as it is streamed across the PCIe bus. This mode offers the best latency.
2. Store-and-forward: The frame is buffered on the adapter before transmitting onto the network.
3. Store-and-forward with poison disabled: As for (2), except that it is guaranteed that frames are never poisoned. When this mode is enabled on any VI, all VIs are placed into store-and-forward mode.

Due to differences in hardware architecture, CTPIO is not available on SFN8000-series or earlier adapters.

## 4.6.1 Underrun, poisoning and fallback

When using cut-through mode, if the frame is not streamed to the adapter at at least line rate, then the frame is likely to be poisoned. This is most likely to happen if the application thread is interrupted while writing the frame to the adapter. In the underrun case, the frame is terminated with an invalid FCS – this is referred to as "poisoning" – and so will be discarded by the link partner. Cut-through mode is currently expected to perform well only on 10G links.

CTPIO may be aborted for other reasons, including timeout while writing a frame, contention between threads using CTPIO at the same time, and the CPU writing data to the adapter in the wrong order.

In all of the above failure cases the adapter falls-back to sending via the traditional DMA mechanism, which incurs a latency penalty. So a valid copy of the packet is always transmitted, whether the CTPIO operation succeeds or not.

After a non-CTPIO send (including DMA fallback), the TX queue must empty before a CTPIO can again succeed.

Normally only an underrun in cut-through mode will result in a poisoned frame being transmitted. In rare cases it is also possible for a poisoned frame to be emitted in store-and-forward mode. If it is necessary to strictly prevent poisoned packets from reaching the network, then poisoning can be disabled globally.

## 4.6.2 CTPIO diagnostics

The adapter maintains counters that show whether CTPIO is being used, and any reasons for CTPIO sends failing. These can be inspected as follows:

```
ethtool -S ethX | grep ctpio
```

Note that some of these counters are maintained on a per-adapter basis, whereas others are per network port.

## 4.7 Filters

*Filters* select which packets are delivered to a virtual interface. Packets that are not selected are ignored and allowed to pass on to the kernel.

Each filter specifies the characteristics of packets for selection. These characteristics are typically packet header fields, including Ethernet MAC address, VLAN tags, IP addresses and port numbers.

A selected packet can be:

- Stolen: the packet is delivered to the virtual interface, but not to the kernel stack.
- Replicated: a copy is delivered to the virtual interface, and might also be delivered to other consumers. Used for multicast packets.
- Sniffed: the packet is delivered to the virtual interface, and to the kernel stack.

### Note

The set of header fields and filter modes that are available vary between adapter model and firmware variant.

### 4.7.1 Multiple Filters

An ef\_vi application can set multiple types of filters on the same virtual interface. Setting an invalid filter or combination of filters causes an error.

## 4.8 Virtual LANs

Ef\_vi only has limited support for *Virtual LANs* (VLANs). This is because ef\_vi operates at the Ethernet frame level, whereas VLANs are usually handled at a higher level:

- Received packets can be filtered by VLAN, but this requires a recent adapter running full feature firmware. There are also limitations on what other filters can simultaneously be used. For more details, see [ef\\_filter\\_spec\\_set\\_vlan\(\)](#).
- Transmitted packets can have their VLAN set by adding the desired VLAN tag to the extended header. Unlike checksums, ef\_vi does not provide an offload for this.

## 4.9 TX Alternatives

*TX alternatives* is a feature available on Solarflare SFN8000 series adapters to provide multiple alternative queues for transmission, that can be used to minimize latency. Different possible responses can be pushed through the TX path on the NIC, and held in different queues ready to transmit. When it is decided which response to transmit, the appropriate alternative queue is selected, and the queued packets are sent. Because the packets are already prepared, and are held close to the wire, latency is greatly reduced.

## Chapter 5

# Example Applications

Solarflare ef\_vi comes with a range of example applications - including source code and make files. This is a quick guide to using them, both for testing ef\_vi's effectiveness in an environment, and as starting points for developing applications.

Most of these applications have additional options to test physical addressing mode, or hardware timestamping. Run with "--help" to check this.

Application	Description
<a href="#">eflatency</a>	Measure latency by pinging a simple message between two interfaces.
<a href="#">efsend</a>	Send UDP packets on a specified interface.
<a href="#">efsink</a>	Receive streams of packets on a single interface.
<a href="#">efforward</a>	Forward packets between two interfaces without modification.
<a href="#">efsend_timestamping</a>	Send UDP packets on a specified interface, with TX timestamping.
<a href="#">efsend_pio</a>	Send UDP packets on a specified interface using Programmed I/O.
<a href="#">efsend_pio_warm</a>	Send UDP packets on a specified interface using Programmed I/O with transmit warming.
<a href="#">efsink_packed</a>	Receive streams of packets on a single interface using packed streams.
<a href="#">efforward_packed</a>	Forward packets between two interfaces without modification using packed stream mode for receive.
<a href="#">efrss</a>	Forward packets between two interfaces without modification, spreading the load over multiple virtual interfaces and threads.
<a href="#">efjumbox</a>	Receive jumbo packets.
<a href="#">exchange</a>	Simplified electronic trading exchange.
<a href="#">trader_onload_ds_efvi</a>	Simplified electronic trader.
<a href="#">efrink_controller</a>	Receive packets on a single interface into a shared memory ring.
<a href="#">efrink_consumer</a>	Consume packets from a shared memory ring.

### 5.1 eflatency

The eflatency application echoes a single packet back and forth repeatedly, measuring the round-trip time.

This is the most useful example application for testing lowest possible latency. It is not a very good sample for building an application, because:

- it uses only one filter
- it operates knowing that there is only ever a single packet on the wire, and so:
  - does not need to refill the rings
  - does not handle multiple event types.

### 5.1.1 Usage

**Server:** `eflatency pong interface`

**Client:** `eflatency ping interface`

where:

- *interface* is the multicast interface on the server or client machine (e.g. `eth0`)

There are various additional options. See the help text for details.

## 5.2 efsend

The `efsend` application sends UDP packets on a specified interface.

The application sends a UDP packet, waits for transmission of the packet to finish and then sends the next.

The number of packets sent, the size of the packet, the amount of time to wait between sends can be controlled.

## 5.3 efsink

The `efsink` application is a demonstration of capturing packets, and the flexibility of filters.

It supports all filter types that `ef_vi` supports. By default it just reports the amount of data captured, but it also demonstrates simple actions upon the packet data, with the option to hexdump incoming packets.

It is a very useful jumping off point as it shows:

- creation of a virtual interface
- creation and installation of filters
- polling the event queue.

### 5.3.1 Usage

To receive a single multicast group:

```
efsink_local interface_ udp:<multicast-addr>:port
```

To receive multiple multicast groups:

```
efsink_interface_ udp:<multicast-addr>:port udp:<multicast-group>:port
```

To receive all multicast traffic:

```
efsink_interface_ multicast-all
```

The efsink application does not send packets.

## 5.4 efforward

The efforward application listens for traffic on one interface and echoes it out of a second; and vice versa. It demonstrates a very simple high-performance bridge.

Some route configuration on the clients might be necessary to get this working, but it is a very simple example, and is very easy to start adding packet re-writing rules etc.

Although this is a viable starting point for a bridging application, a better option might be the SolarCapture API, which includes a more complete pre-made bridging application.

## 5.5 efsend\_timestamping

The efsend\_timestamping application sends UDP packets on a specified interface.

The application sends a UDP packet, waits for transmission of the packet to finish and then sends the next.

This application requests tx timestamping, allowing it to report the time each packet was transmitted.

The number of packets sent, the size of the packet, the amount of time to wait between sends can be controlled.

## 5.6 efsend\_pio

The efsend\_pio application sends UDP packets on a specified interface.

Packet data is copied to the NIC's PIO buffer before being sent, which typically results in lower latency sends compared to accessing packet data stored on the host via DMA, which is the method used by the efsend sample app.

The application sends a UDP packet, waits for transmission of the packet to finish and then sends the next.

The number of packets sent, the size of the packet, the amount of time to wait between sends can be controlled.

## 5.7 efsend\_pio\_warm

The efsend\_pio\_warm application demonstrates PIO transmit warming.

The application sends a UDP packet using PIO each time a trigger fires. While waiting for a trigger, the application can warm the PIO transmit path to reduce latency of the subsequent send.

The effect of warming can be assessed by measuring the time from when the trigger fires to when the corresponding packet leaves the adapter.

Several parameters can be controlled including the delay between triggers, the enablement of warming and the frequency of warming while waiting for a trigger.

## 5.8 efsink\_packed

The efsink\_packed application is a variant of [efsink](#) that demonstrates usage of the packed-stream firmware.

## 5.9 efforward\_packed

The efforward\_packed application is a variant of [efforward](#) that demonstrates usage of the packed-stream firmware.

## 5.10 efrss

The efrss application is a variant of [efforward](#). It demonstrates automatically spreading the load over multiple threads, using a vi\_set and RSS.

## 5.11 efdelegated\_client

The efdelegated\_client application demonstrates usage of OpenOnload's "Delegated Sends" feature. This API allows you to do delegate TCP sends for a particular socket to some other mechanism. For example, this sample uses the ef\_vi layer-2 API in order to get lower latency than is possible with a normal send() call.

The API essentially boils down to first retrieving the packet headers, adding your own payload to form a raw packet, sending the packet and finally telling Onload what it was you sent so it can update the internal TCP state of the socket.

This sample application allows you to compare the performance of normal sends using the kernel stack, using OpenOnload and using ef\_vi with the delegated sends API. It establishes a TCP connection to the server process, which starts sending UDP multicast messages. The client receives these messages, and replies to a subset of them with a TCP message. The server measures the latency from the UDP send to the TCP receive.



## 5.12 efjumborx

The efjumborx application demonstrates receiving jumbo packets.

Various parameters can be set, including the interface, IP address and port to use. Verbose output can be enabled to view progress. The received packet can be printed as text or in hexadecimal.

## 5.13 exchange

The exchange application plays the role of a simplified electronic trading exchange. It is to be used in conjunction with the [trader\\_onload\\_ds\\_efvi](#) application.

## 5.14 trader\_onload\_ds\_efvi

The trader\_onload\_ds\_efvi application demonstrates various techniques to reduce latency. These techniques are often useful in electronic trading applications, and so this example takes the form of an extremely simplified electronic trading system.

The [exchange](#) application provides a simplified electronic trading exchange, and this application provides a simplified electronic trader.

A trader\_onload\_ds\_efvi application demonstrates similar techniques for Onload.

For full details, see the `README` file in the `tests/trade_sim` directory.

### 5.14.1 Usage

For normal socket-based sends, run as follows:

**Server:** `onload -p latency-best ./exchange mcast-intf`

**Client:** `onload -p latency-best ./trader_onload_ds_efvi mcast-intf server`

For "delegated" sends, run as follows:

**Server:** `onload -p latency-best ./exchange mcast-intf`

**Client:** `onload -p latency-best ./trader_onload_ds_efvi -d mcast-intf server`

where:

- `mcast-intf` is the multicast interface on the server or client machine (e.g. `eth0`)
- `server` is the IP address of the server machine (e.g. `192.168.0.10`)

There are various additional options. See the help text for details.

## 5.15 efrink\_controller

The efrink\_controller application receives streams of packets on a single interface into a shared memory ring. This can be read by multiple consumers, such as [efrink\\_consumer](#).

This is an advanced technique which has the potential to provide a more efficient way to receive data when multiple consumer processes want to receive the same data stream(s) from the network.

A single controller sets up a region of shared memory, and posts chunks of this to the RX ring of the NIC. As the NIC writes packet data, it returns completion events. The controller receives these events and marks each packet buffer as complete.

One or more readers such as [efrink\\_consumer](#) can access the shared memory to read the packets. The data is protected via a generation counter.

## 5.16 efrink\_consumer

The efrink\_consumer application consumes packets from a shared memory ring, that is being managed by [efrink\\_controller](#).

Multiple copies of efrink\_consumer can run at the same time. For best performance, all processes should share the same NUMA node/cache.

## 5.17 Building the Example Applications

The ef\_vi example applications are built along with the Onload installation and will be present in the /Onload-<version>/build/gnu\_x86\_64/tests/ef\_vi subdirectory. In the build directory there will be gnu, gnu\_x86\_64, x86\_64\_linux-<kernel version> directories:

- files under the gnu directory are 32-bit (if these are built)
- files under the gnu\_x86\_64 directory are 64-bit.

Source code files for the example applications exist in the /Onload-<version>/src/tests/ef\_vi subdirectory.

After running the onload\_install command, example applications exist in the /Onload-<version>/build/gnu\_x86\_64/tests/ef\_vi subdirectory.

To rebuild the example applications you must have the Onload-<version>/scripts subdirectory in your path and use the following procedure:

```
[root@server01 Onload-<version>]# cd scripts/
[root@server01 scripts]# export PATH="$PWD:$PATH"
[root@server01 scripts]# cd ../build/gnu_x86_64/tests/ef_vi/
[root@server01 ef_vi]# make clean
[root@server01 ef_vi]# make
```

## Chapter 6

# Using ef\_vi

This part of the documentation gives information on using ef\_vi to write and build applications.

### 6.1 Components

All components required to build and link a user application with the Solarflare ef\_vi API are distributed with Onload. When Onload is installed all required directories/files are located under the Onload distribution directory:

### 6.2 Compiling and Linking

Applications or libraries using ef\_vi will need to include the header files in `src/include/etherfabric/`

The application will need to be linked with `libciul1.a` or `libciul.so`, which can be found under the "build" directory after running `scripts/onload_build` or `scripts/onload_install`.

If compiling your application against one version of Onload, and running on a system with a different version of Onload, some care is required. Onload currently preserves compatibility and provides a stable API between the ef\_vi user-space and the kernel drivers, so that applications compiled using an older ef\_vi library will work when run with newer drivers. Compatibility in the other direction (newer ef\_vi libraries running with older drivers) is not guaranteed. Finally, Onload does not currently maintain compatibility between compiling against one version of the ef\_vi libraries, and then running against another.

The simplest approach is to link statically to `libciul`, as this ensures that the version of the library used will match the one you have compiled against. If linking dynamically, it is recommended that you keep `libciul.so` and the application binary together. `onload_install` does not install `libciul.so` into system directories to avoid the installed version being used in place of the version you compiled against.

For those wishing to use ef\_vi in combination with Onload there should be no problem linking statically to `libciul` and dynamically to the other libraries to allow the Onload intercepts to take effect. The `ef_delegated` example application does exactly this.

## 6.3 Setup

Applications requiring specific features can check the versions of software:

- use `ef_vi_version_str()` to get the version of ef\_vi
- use `ef_vi_driver_interface_str()` to get the char driver interface required by this build of ef\_vi.

Applications can also check for specific capabilities:

- use `ef_vi_capabilities_get()` to get the value of a given capability
- use `ef_vi_capabilities_max()` to get the number of available capabilities
- use `ef_vi_capabilities_name()` to get a human-readable string describing a given capability.

Users of ef\_vi must do the following to setup:

1. Obtain a driver handle by calling `ef_driver_open()`.
2. Allocate a protection domain by calling one of the following:
  - `ef_pd_alloc()`
  - `ef_pd_alloc_by_name()`
  - `ef_pd_alloc_with_vport()`.
3. Allocate a virtual interface (VI), encapsulated by the type `ef_vi`, by calling `ef_vi_alloc_from_pd()`.

```
/* Allocate a protection domain */
int ef_pd_alloc(ef_pd *pd,
               ef_driver_handle pd_dh,
               int ifindex,
               enum ef_pd_flags flags);

int ef_pd_alloc_by_name(ef_pd *pd,
                       ef_driver_handle pd_dh,
                       const char* cluster_or_intf_name,
                       enum ef_pd_flags flags);

/* Get the interface for a protection domain. */
const char* ef_pd_interface_name(ef_pd *pd);
```

### Figure: Create a Protection Domain

```
/* Allocate a virtual interface */
int ef_vi_alloc_from_pd(ef_vi *vi, ef_driver_handle vi_dh,
                       ef_pd *pd, ef_driver_handle pd_dh,
                       int eventq_cap, int rxq_cap, int txq_cap,
                       ef_vi *opt_evq, ef_driver_handle opt_evq_dh,
                       enum ef_vi_flags flags);
```

### Figure: Allocate a Virtual Interface

### 6.3.1 Using Virtual Interface Sets.

A virtual interface set can be used instead of a single virtual interface, to distribute load using RSS. Functionality is almost the same as working with a single virtual interface:

- To allocate the virtual interfaces:
  - use `ef_vi_set_alloc_from_pd()` to allocate the set
  - then use `ef_vi_alloc_from_set()` to allocate each virtual interface in the set
- To add a filter:
  - use `ef_vi_set_filter_add()` to add the filter onto the set, rather than adding it to each virtual interface individually (which would cause replication on a 7000-series NIC).

The `efrss` sample gives an example of usage.

## 6.4 Creating packet buffers

Memory used for packet buffers is allocated using standard functions such as `posix_memalign()`. A packet buffer should be at least as large as the value returned from `ef_vi_receive_buffer_len()`.

The packet buffers must be pinned so that they cannot be paged, and they must be registered for DMA with the network adapter. These requirements are enforced by calling `ef_memreg_alloc()` to register the allocated memory for use with `ef_vi`.

The type `ef_iovec` encapsulates a set of buffers. The adapter uses a special address space to identify locations in these buffers, and such addresses are designated by the type `ef_addr`.

```
/* Allocate a memory region */
int ef_memreg_alloc(ef_memreg* mr, ef_driver_handle mr_dh,
                  ef_pd* pd, ef_driver_handle pd_dh,
                  void* p_mem, int len_bytes);
```

### Figure: Allocate a Memory Region

To improve performance, registered memory regions for packet buffers should be aligned on (minimum) 4KB boundaries for regular pages or 2MB boundaries when using huge pages, and should be contiguous.

#### 6.4.1 Buffer Tables

The memory for packet buffers is mapped using a buffer table. Implementation of this can differ, but typically:

- The buffer table is allocated from memory that is also used for other things, so can vary in size
- Entries in this table are split into sets of 32 entries
- In the default configuration there are typically 1926 sets available, giving a maximum of 61632 entries
- Each entry maps a chunk of naturally-aligned contiguous memory:

- 4KB on earlier NICs
- 4KB, 64KB, 1MB or 4MB on 7000-series and later NICs
- Each packet buffer consumes 2KB.

The total number of packet buffers available might be fewer than expected from the above:

- Some sets of entries might not be fully utilized.  
In particular, the entries in a set must have the same owner ID, and so where there are multiple owner IDs, there will be unused holes in the table.
- Setting up virtual interfaces can consume buffer table resources.  
On 5000- and 6000-series NICs, every page of host memory that is allocated to back a TX queue, RX queue or event queue uses a buffer table entry to map it. So a few hundred virtual interfaces will consume a lot of entries, especially if they are set up with large event queues.
- The descriptor cache can reduce buffer table resources.  
On 7000-series and later NICs, the descriptor cache consumes memory on the NIC that is otherwise available for the buffer table.

## 6.5 Transmitting Packets

To transmit packets, the basic process is:

1. Write the packet contents (including all headers) into one or more packet buffers.  
The packet buffer memory must have been previously registered with the protection domain.
2. Post a descriptor for the filled packet buffer onto the TX descriptor ring, by calling `ef_vi_transmit_init()` and `ef_vi_transmit_push()`, or `ef_vi_transmit()`.  
A doorbell is "rung" to inform the adapter that the transmit ring is non-empty. If the transmit descriptor ring is empty when the doorbell is rung, 'TX PUSH' is used. In 'TX\_PUSH', the doorbell is rung and the address of the packet buffer is written in one shot improving latency. TX\_PUSH can cause ef\_vi to poll for events, to check if the transmit descriptor ring is empty, before sending which can lead to a latency versus throughput trade off in some scenarios.
3. Poll the event queue to find out when the transmission is complete.  
See [Handling Events](#).  
When transmitting, polling the event queue is less critical; but does still need to be done. The events of interest are `EF_EVENT_TYPE_TX` or `EF_EVENT_TYPE_TX_WITH_TIMESTAMP` telling you that a transmit completed, and `EF_EVENT_TYPE_TX_ERROR` telling you that a transmit failed. `EF_EVENT_TX_Q_ID()` can be used to extract the id of the referenced packet, or you can just rely on the fact that ef\_vi always transmits packets in the order they are submitted.
4. Handle the resulting event.  
Reclaim the packet buffer for re-use.

```
// construct packet with proper headers
// Post on the transmit ring
ef_vi_transmit_init(&vi, addr, len, id);
// Ring doorbell
ef_vi_transmit_push(&vi);
```

**Figure: Transmit Packets**

### 6.5.1 Transmitting Jumbo Frames

Packets of a size smaller than the interface MTU but larger than the packet buffer size must be sent from multiple buffers as jumbo packets. A single `EF_EVENT_TYPE_TX` (or `EF_EVENT_TYPE_TX_ERROR`) event is raised for the entire transmit:

- Use `ef_vi_transmitv()` to chain together multiple segments with a higher total length.
- MTU is not enforced. If the transmit is to remain within MTU, the application must check and enforce this.
- The segments must at least split along natural (4k packet) boundaries, but smaller segments can be used if desired..

### 6.5.2 Programmed I/O

Programmed IO is usable only on 7000-series cards, not on the older cards. It allows for faster transmit, especially of small packets, but the hardware resources available for it are limited.

For this reason, a PIO buffer must be explicitly allocated and associated with a virtual interface before use, by calling `ef_pio_link_vi()`.

Data is copied into the PIO buffer with `ef_pio_memcpy()`.

When the PIO buffer is no longer required it should be unlinked by calling `ef_pio_unlink_vi()`, and then freed by calling `ef_pio_free()`.

### 6.5.3 Cut-through PIO

A complete example showing the use of CTPIO with ef\_vi is given in `Onload-<version>/src/tests/rtt/rtt_efvi.c`.

#### Note

CTPIO bypasses the main adapter datapath, and as a result does not support checksum offload. Frames sent with CTPIO are placed on the wire without modification, so checksum fields must be completed in software before sending.

Here is the sequence of steps needed:

1. When allocating a VI (`ef_vi_alloc_from_pd()`) set the `EF_VI_TX_CTPIO` flag.
2. To initiate a send, form a complete Ethernet frame (including L3/L4 checksums, but excluding FCS) in host memory. Initiate the send with `ef_vi_transmit_ctpio()` or `ef_vi_transmitv_ctpio()`.
3. Post a fall-back descriptor using `ef_vi_transmit_ctpio_fallback()` or `ef_vi_transmitv_ctpio_fallback()`. These calls are used just like the standard DMA send calls (`ef_vi_transmit()` etc.), and so must be provided with a copy of the frame in registered memory.

The posting of a fall-back descriptor is not on the latency critical path, provided the CTPIO operation succeeds. However, it must be posted before posting any further sends on the same VI.

## 6.5.4 TX Alternatives

TX alternatives is a features available on Solarflare SFN8000 series adapters. To use TX alternatives for a given virtual interface, you must set the EF\_VI\_TX\_ALT flag when you allocate the virtual interface. You must then allocate a set of TX alternatives for the virtual interface by calling `ef_vi_transmit_alt_alloc()`.

```
int ef_vi_transmit_alt_alloc(struct ef_vi* vi,
    ef_driver_handle vi_dh,
    int num_alts, int buf_space);
```

The number of virtual interfaces that can use TX alternatives simultaneously is limited, and varies by adapter and port mode. Typical limitations are as follows:

- SFN8522, 2x10Gb: at least 6 virtual interfaces can use TX alternatives
- SFN8542, 2x40Gb: at least 6 virtual interfaces can use TX alternatives
- SFN8542, 1x40Gb + 2x10Gb: at least 3 virtual interfaces can use TX alternatives
- SFN8542, 4x10Gb: TX alternatives are *not* available.

This creates a set of TX alternatives. The TX alternatives remain allocated until the virtual interface is freed. They are given sequential IDs, from 0 upwards.

You can get the number of TX alternatives in a set by calling `ef_vi_transmit_alt_num_ids()`.

```
unsigned ef_vi_transmit_alt_num_ids(ef_vi* vi);
```

You can then buffer responses on the different TX alternatives, choose which to transmit, and discard any that are no longer required:

- All TX alternatives are initially in the STOP state, and any packets sent to them are buffered.
- To send packets to a particular TX alternative, select the TX alternative by calling `ef_vi_transmit_alt_select()`, and then send the packets to the virtual interface using normal send calls such as `ef_vi_transmit()`.
- To transmit the packets that are buffered on a TX alternative, call `ef_vi_transmit_alt_go()`.  
This transitions the TX alternative to the GO state. While the TX alternative remains in this state, any further packets sent to it are transmitted immediately.
- To transition the TX alternative back to the STOP state, call `ef_vi_transmit_alt_stop()`.
- To discard the packets buffered on a TX alternative, transition it to the DISCARD state by calling `ef_vi_transmit_alt_discard()`.

```
int ef_vi_transmit_alt_select(struct ef_vi* vi, unsigned alt_id);
int ef_vi_transmit_alt_stop(struct ef_vi* vi, unsigned alt_id);
int ef_vi_transmit_alt_go(struct ef_vi* vi, unsigned alt_id);
int ef_vi_transmit_alt_discard(struct ef_vi* vi, unsigned alt_id);
```



As packets are transmitted or discarded, events of type `EF_EVENT_TYPE_TX_ALT` are returned to your application. Your application should normally wait until all packets have been processed before transitioning to a different state.

Calls are available to calculate the buffer usage:

- To query the amount of user-visible buffering that will be available, call `ef_vi_transmit_alt_query_buffering()`
- To query the per-packet overheads, call `ef_vi_transmit_alt_query_overhead()`.
- To calculate a packet's buffer usage, pass the per-packet overheads and the packet length to `ef_vi_transmit_alt_usage()`.

```
int ef_vi_transmit_alt_query_buffering(struct
    ef_vi* vi, int ifindex,
                                     ef_driver_handle vi_dh, int n_alts);

int
ef_vi_transmit_alt_query_overhead(ef_vi* vi,
    struct ef_vi_transmit_alt_overhead* params);

uint32_t
ef_vi_transmit_alt_usage(const struct
    ef_vi_transmit_alt_overhead* params,
    uint32_t pkt_len);
```

## 6.6 Handling Events

The event queue is a channel from the adapter to software which notifies software when packets arrive from the network, and when transmits complete (so that the buffers can be freed or reused). Application threads retrieve these events in one of the following ways:

- A thread can busy-wait for an event notification by calling `ef_eventq_poll()` repeatedly in a tight loop. This gives the lowest latency.
- A thread can block until event notifications arrive (or a timeout expires) by calling `ef_eventq_wait()`. This frees the CPU for other usage.

The batch size for polling must be at least `EF_VI_EVENT_POLL_MIN_EVTS`. It should be greater than the batch size for refilling to detect when the receive descriptor ring is going empty.

When timestamping is enabled, a number of timestamps per second are added to the event queue, even when no packets are being received. It is important that the application regularly polls the VI event queue, to avoid an event queue overflow (`EF_EVENT_TYPE_OFLOW`) which can result in an undetermined state for the VI.

### 6.6.1 Blocking on a file descriptor

Ef\_vi supports integration with other types of I/O via the select, poll and epoll interfaces. Each virtual interface is associated with a file descriptor. The ef\_vi layer supports blocking on a file descriptor until it has events ready, when it becomes readable. This feature provides the functionality that is already provided by `ef_eventq_wait()` with the added benefit that as you are blocking on a file descriptor, you can block for events on a virtual interface along with other file descriptors at the same time.

The file descriptor to use for blocking is the driver handle that was used to allocate the virtual interface.

Before you can block on the file descriptor, you need to prime interrupts on the virtual interface. This is done by calling `ef_vi_prime()`.

```
int ef_vi_prime(ef_vi* vi, ef_driver_handle dh,
               unsigned current_ptr);
```

When this function is called, you must tell it how many events you've read from the eventq which can be retrieved by using `ef_eventq_current()`. Then you can simply block on the file descriptor becoming readable by using `select()`, `poll()`, `epoll()`, etc. When the file descriptor is returned as readable, you can then get the associated events by polling the eventq in the normal way. Note that at this point, you must call `ef_vi_prime()` again (with the current value from `ef_eventq_current()`) before blocking on the file descriptor again.

The `efsink` example code offers a simple example.

## 6.7 Receiving packets

To receive packets, the basic process is:

1. Post descriptors for empty packet buffers onto the RX descriptor ring, by calling `ef_vi_receive_init()` and `ef_vi_receive_push()`, or `ef_vi_receive_post()`.

Receive descriptors should be posted in multiples of 8. When an application pushes 10 descriptors, `ef_vi` will push 8 and `ef_vi` will ignore descriptor batch sizes < 8. Users should beware that if the ring is empty and the application pushes < 8 descriptors before blocking on the event queue, the application will remain blocked as there are no descriptors available to receive packets so nothing gets posted to the event queue.

Posting descriptors should ideally be done in batches, by a thread that is not on the critical path. A small batch size means that the ring is kept more full, but a large batch size is more efficient. A size of 8, 16 or 32 is probably the best compromise.

2. Poll the event queue to see that they are now filled.

See [Handling Events](#).

Packets are written into the buffers in FIFO order.

3. Handle the resulting event and the incoming packet.

Since the adapter is cut-through, errors in receiving packets like multicast mismatch, CRC errors, etc. are delivered along with the packet. The software must detect these errors and recycle the associated packet buffers.

### 6.7.1 Finding the Packet Data

Use the queue ID to find the packet and its buffer. The diagram below shows the layout of the buffer, and the functions and macros for accessing it:

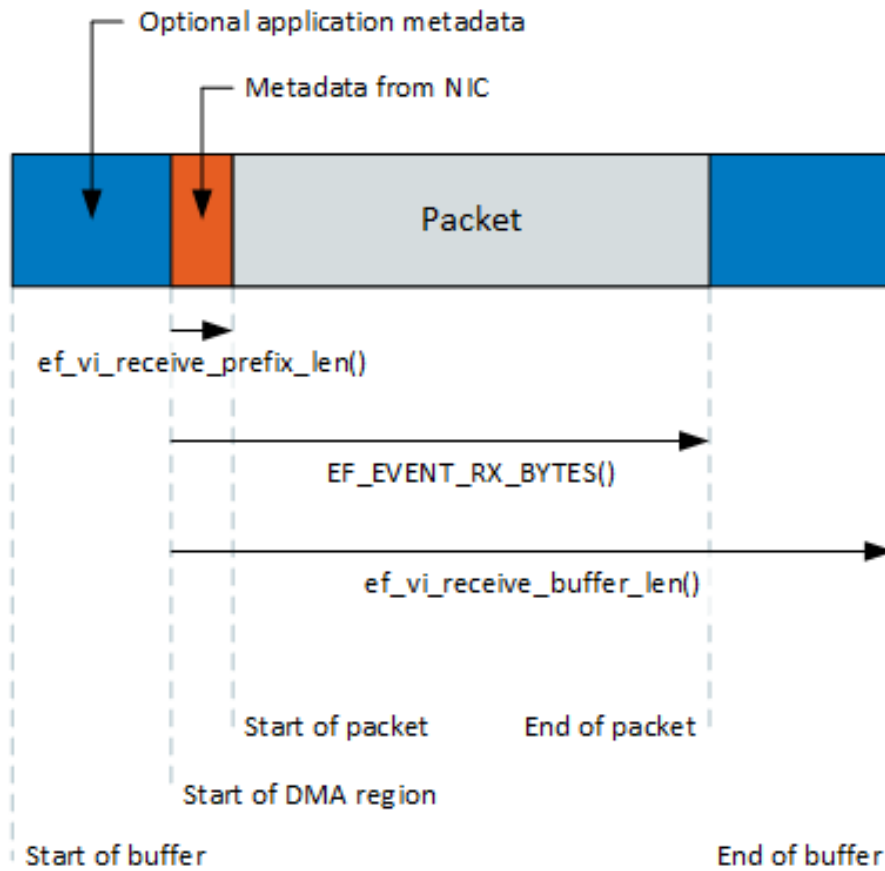


Figure 6.1 Layout and functions for a packet buffer

- Use `ef_vi_receive_prefix_len()` to find the offset for the packet data. (Remember that this includes the headers, as ef\_vi does not do any protocol handling.)
- `EF_EVENT_RX_BYTES()` gives the number of bytes received.
- `ef_vi_receive_query_layout()` gives more information.

Packets that are smaller than the packet buffer size are delivered one per packet buffer. An event is raised for each packet buffer that is filled:

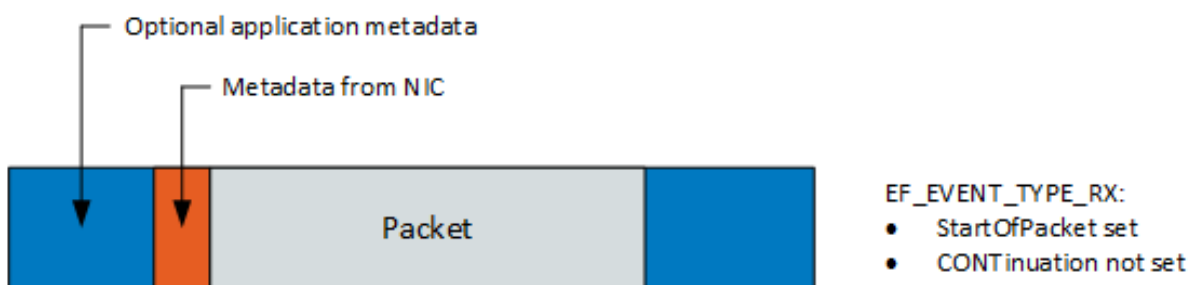


Figure 6.2 A single (non-jumbo) packet

## 6.7.2 Receiving Jumbo Packets

Packets of a size smaller than the interface MTU but larger than the packet buffer size are delivered in multiple buffers as jumbo packets. An event is raised for each packet buffer that is filled. For example, this is a packet that requires two buffers:

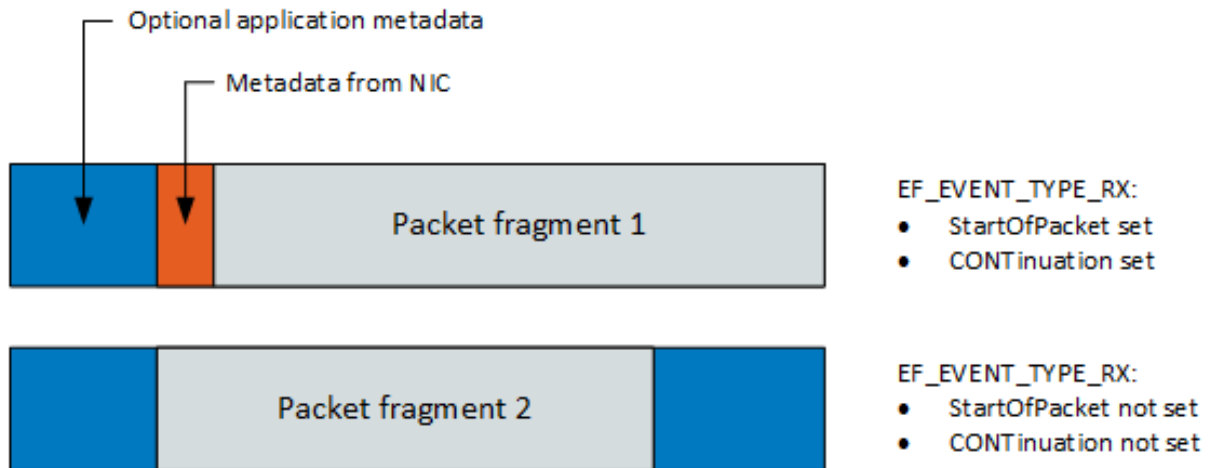


Figure 6.3 A 2-buffer jumbo packet

and this is a larger packet that requires  $N$  buffers:

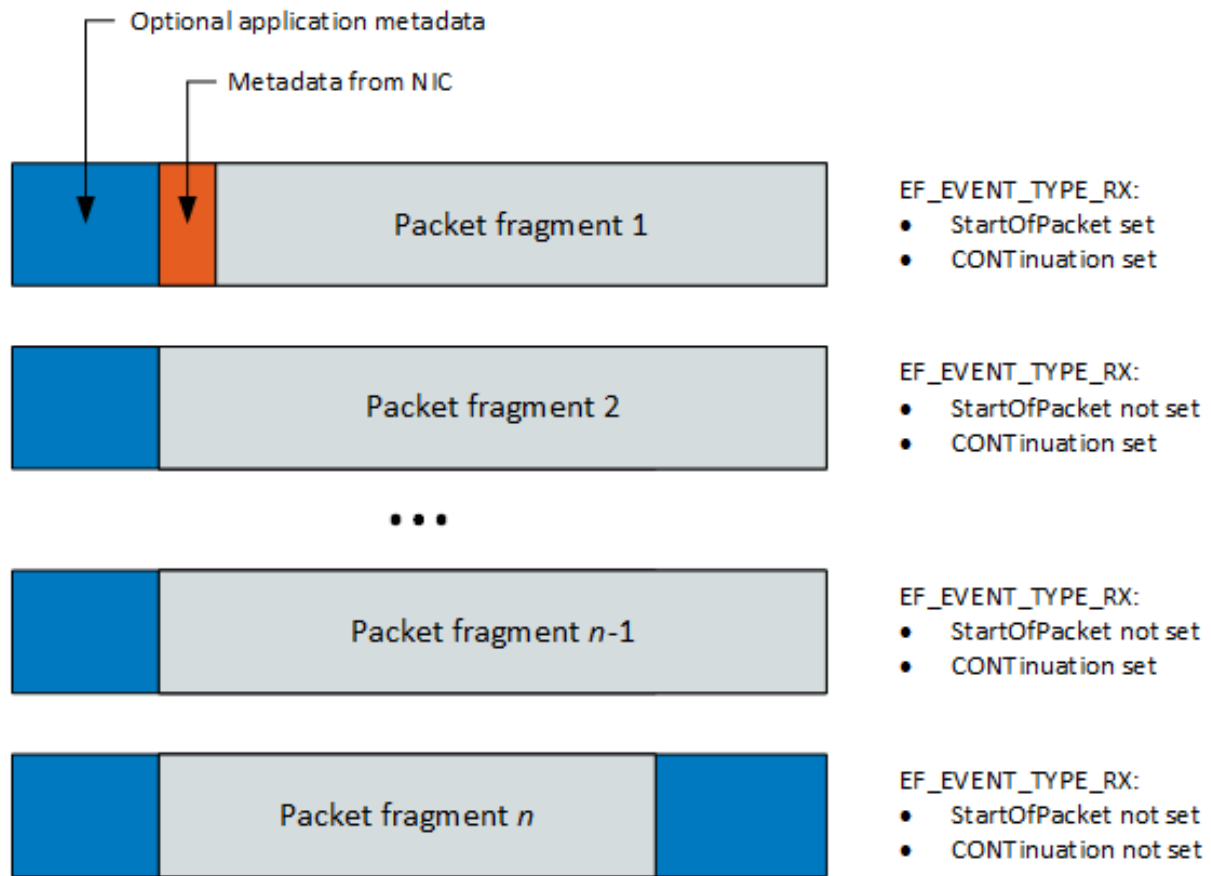


Figure 6.4 An N-buffer jumbo packet

- The `EF_EVENT_RX_CONT()` macro can be used to check if this is not the last part of the packet, and that the next receive (on this RX descriptor ring) should also be examined as being part of this jumbo frame.
- The `EF_EVENT_RX_BYTES()` macro gets the number of bytes of the packet that have been received. This is a cumulative total, so the value for the last part of the frame is the total packet length.
- If `EF_EVENT_RX_DISCARD_TRUNC` is set, this indicates that the packet buffer has been dropped, and so the jumbo packet has been truncated. But there might still be more parts of the jumbo packet that arrive after the drop. All packet buffers should be discarded until one is received with `EF_EVENT_RX_SOP` set, marking the start of a new packet.

### 6.7.3 RX Event Merging

RX event merging can merge multiple events into a single `EF_EVENT_TYPE_RX_MULTI` event:

- There can potentially be multiple packets per event.
- Events can only be merged to a single `EF_EVENT_TYPE_RX_MULTI` event when their flags are the same.
- When an `EF_EVENT_TYPE_RX_MULTI` event occurs, `ef_vi_receive_unbundle()` must always be used to find the individual buffers.

- When an `EF_EVENT_TYPE_RX_MULTI_DISCARD` event occurs, `ef_vi_receive_unbundle()` must also always be called, even if the application is unconcerned with the detailed discard information.
- When the first buffer of a packet is found, `ef_vi_receive_get_bytes()` can then give the length of the packet:

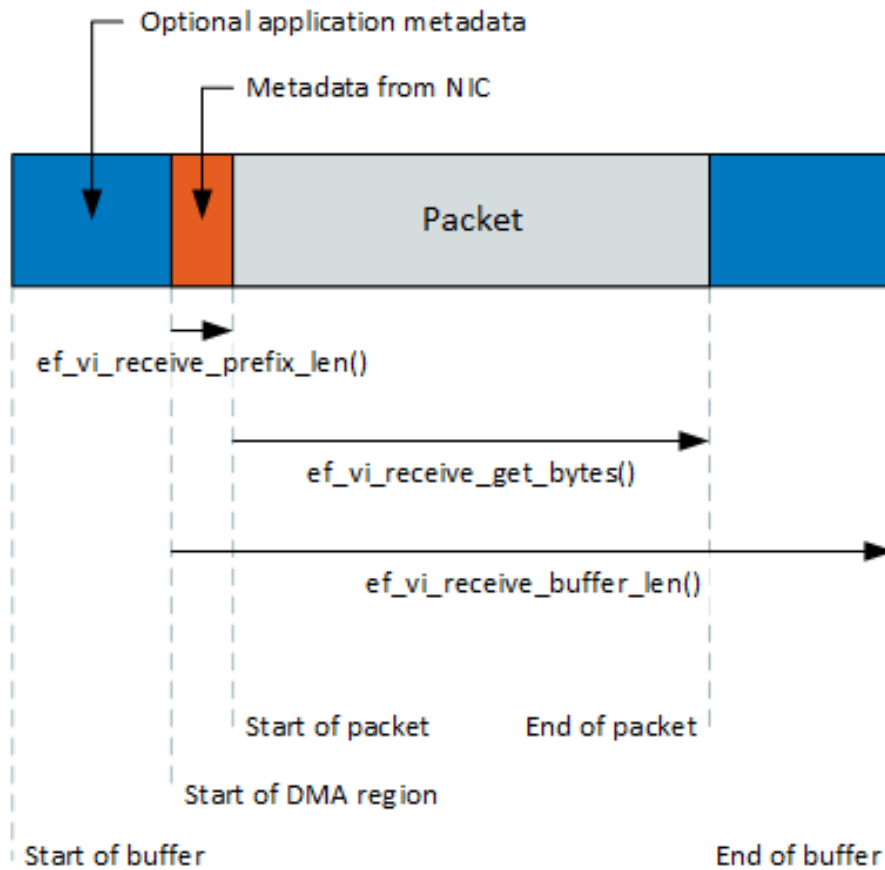


Figure 6.5 Layout and functions for RX event merging

If consecutive single-buffer packets are received, they can be merged:

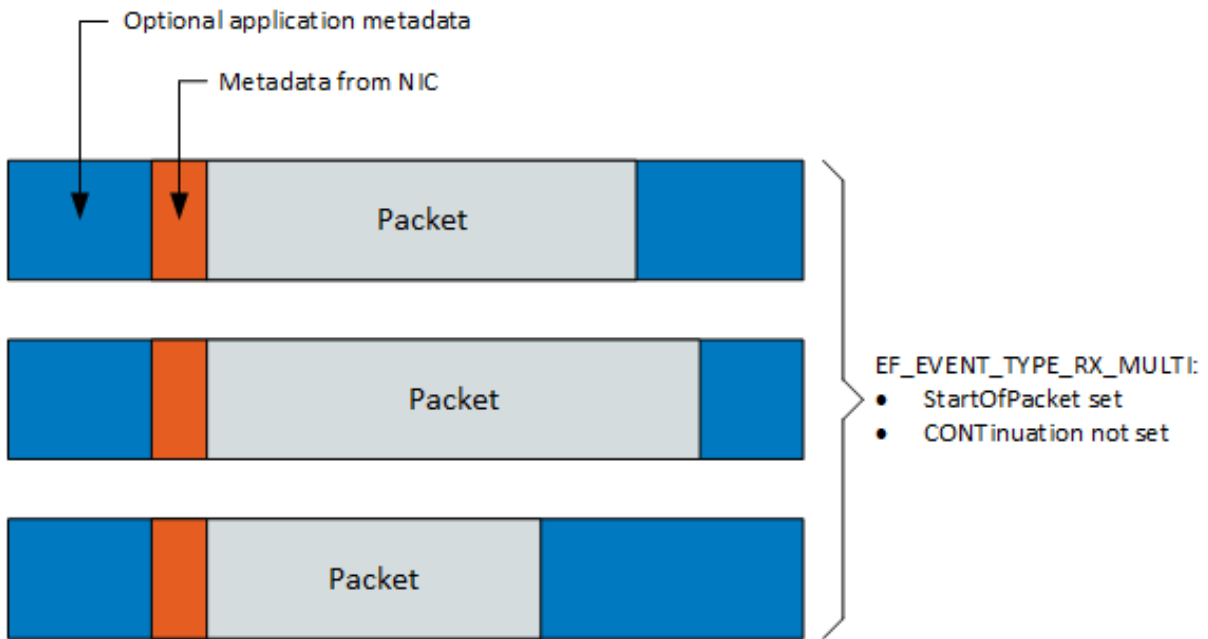


Figure 6.6 RX event merging for non-jumbo packets

For a two-buffer jumbo packet, the events have different flags, and so cannot be merged:

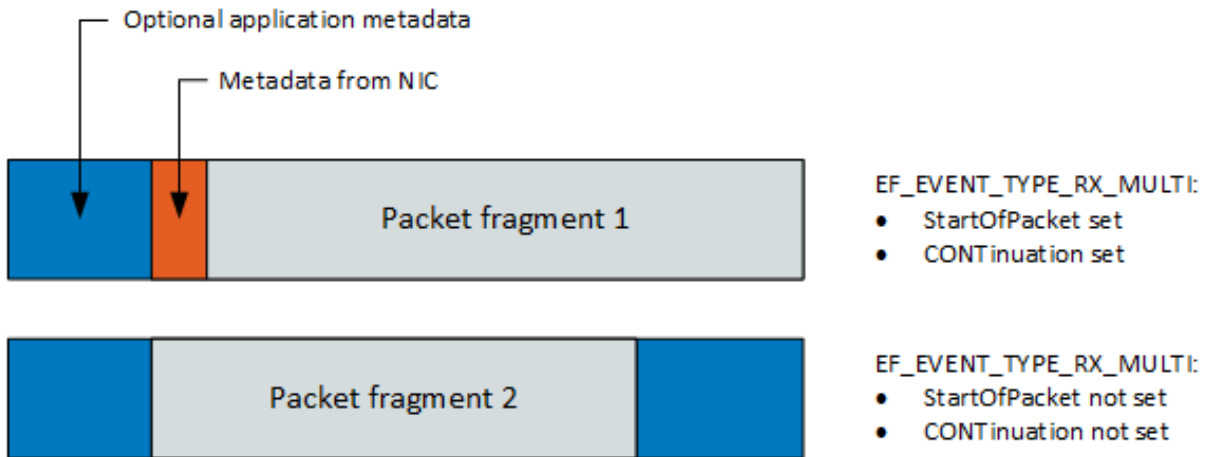


Figure 6.7 RX event merging for a 2-buffer jumbo packet

For a larger  $N$ -buffer jumbo packet, the events for the middle buffers have the same flags, and so can be merged:

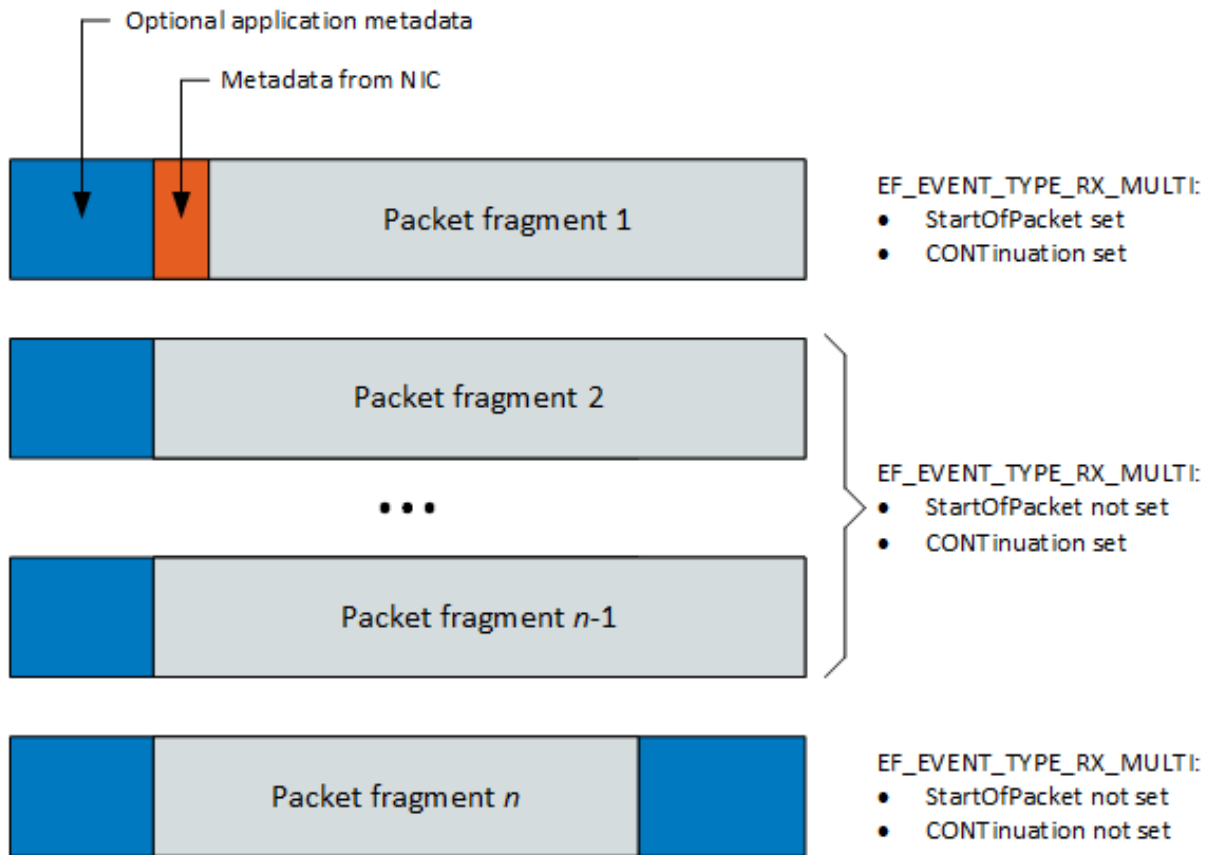


Figure 6.8 Event merging for an  $_N$ -buffer jumbo packet

So for jumbo packets, the following multiple events are typically raised:

1. An event with `EF_EVENT_RX_MULTI_SOP` and `EF_EVENT_RX_MULTI_CONT` containing the first buffer. `ef_vi_receive_get_bytes()` gives the length of the whole jumbo packet.
2. If there are more than two buffers, event(s) with `EF_EVENT_RX_MULTI_CONT` containing the middle buffers.
3. An event with ! `EF_EVENT_RX_MULTI_SOP` and ! `EF_EVENT_RX_MULTI_CONT` containing the final buffer. The length of the payload can be inferred based on the total length of the Jumbo packet minus the lengths of previous payloads.

There is example code in the following locations:

- the `efsink` example application
- the `handle_rx_scatter_merge()` function within `openonload/src/lib/transport/ip/netif_event.c`.



## 6.8 Adding Filters

Filters are the means by which the adapter decides where to deliver packets it receives from the network. By default all packets are delivered to the kernel network stack. Filters are added by an ef\_vi application to direct received packets to a given virtual interface.

- If a filter cannot be added, for example because an incompatible filter already exists, an error is returned.
- By default the 'all' filters are sending everything to the kernel. They are equivalent to setting promiscuous mode on the NIC.
- On SFN5000 and SFN6000 series NICs each filter can only exist for one virtual interface, and so each packet which arrives can be forwarded only to a single application (unless two applications share a stack). Only the first application to insert a specific filter will succeed; other applications will then get an error.  
Note that this includes filters inserted both by other ef\_vi applications and by Onload - which typically uses only fully connected and listen filters.
- The SFN7000 and SFN8000 series NICs are not subject to this restriction. If two applications insert the same filter, a copy of the packets is delivered to each application and applications remain unaware of each other.
- IP filters do not match IP fragments, which are therefore received by the kernel stack. If this is an issue, layer 2 filters should be installed by the user.
- There are no ranges, or other local wildcard support. To filter on a range of values, one of the following is required:
  - insert multiple filters, one per value in the range (NICs support upwards of a thousand filters easily)
  - have the interesting traffic sent to a specific MAC address, and use a MAC address filter
  - have the interesting traffic sent to a specific VLAN, and use a VLAN filter.
- Cookies are used to remove filters.
- De-allocating a virtual interface removes any filters set for the virtual interface.

Filters are checked in the following order, which is roughly most-specific first:

1. Fully connected TCP/UDP. (Specifies local and remote port and IP)
2. Listen socket. (Specifies local port and IP, but allows any remote IP/port)
3. Destination MAC address, and optionally VLAN. (Useful for multicast reception, though IP can be used instead if preferred. Also useful for custom protocols.)
4. Everything else. (All unicast, all multicast.)

```
void ef_filter_spec_init(ef_filter_spec* fs,
                       enum ef_filter_flags flags);

int ef_filter_spec_set_ip4_local(ef_filter_spec* fs,
                                int protocol,
                                unsigned host_be32, int port_be16);

int ef_filter_spec_set_ip4_full(ef_filter_spec* fs,
                                int protocol,
                                unsigned host_be32, int port_be16,
                                unsigned rhost_be32, int rport_be16);

int ef_filter_spec_set_vlan(ef_filter_spec* fs,
                            int vlan_id);
```

```

int ef_filter_spec_set_eth_local(ef_filter_spec* fs,
                               int vlan_id,
                               const void *mac);

int ef_filter_spec_set_unicast_all(ef_filter_spec* fs);

int ef_filter_spec_set_multicast_all(
    ef_filter_spec* fs);

int ef_filter_spec_set_unicast_mismatch(
    ef_filter_spec* fs);

int ef_filter_spec_set_multicast_mismatch(
    ef_filter_spec* fs);

int ef_filter_spec_set_port_sniff(ef_filter_spec* fs,
                                  int promiscuous);

int ef_filter_spec_set_tx_port_sniff(
    ef_filter_spec* fs);

int ef_filter_spec_set_block_kernel(
    ef_filter_spec* fs);

int ef_filter_spec_set_block_kernel_multicast(
    ef_filter_spec* fs);

int ef_filter_spec_set_block_kernel_unicast(
    ef_filter_spec* fs);

int ef_vi_filter_add(ef_vi* vi, ef_driver_handle dh,
                    const ef_filter_spec* fs,
                    ef_filter_cookie* filter_cookie_out);

int ef_vi_filter_del(ef_vi* vi, ef_driver_handle dh,
                    ef_filter_cookie* filter_cookie);

int ef_vi_set_filter_add(ef_vi_set*,
                        ef_driver_handle dh,
                        const ef_filter_spec* fs,
                        ef_filter_cookie* filter_cookie_out);

int ef_vi_set_filter_del(ef_vi_set*,
                        ef_driver_handle dh,
                        ef_filter_cookie* filter_cookie);

```

**Figure: Creating Filters**

## 6.8.1 Filter permission requirements

Super-user rights (specifically CAP\_NET\_ADMIN) are needed to use the following filters:

- IP proto (and the VLAN variant)
- Ethertype (and the VLAN variant)
- Unicast-mismatch (and the VLAN variant)
- Multicast-mismatch (and the VLAN variant)
- Unicast-all
- Multicast-all
- Kernel block
- Sniff.

## 6.8.2 Filters available per Firmware Variant

The kernel will install a destination MAC address filter for each Solarflare interface. The kernel will install a broadcast MAC address filter for each Solarflare interface.

A broadcast MAC address is treated like any other MAC address. Multiple applications can insert the same filter and all will receive a copy of the received traffic.

As a rule, more-specific filters will capture traffic over less-specific filters.

When the Solarflare adapter is configured to use the full\_featured firmware variant, the following combinations of `ef_filter_spec_set_*`() calls are supported, and are applied in this order:

- `ef_filter_spec_set_ip4_full()` or `ef_filter_spec_set_ip6_full()`
- `ef_filter_spec_set_eth_type()`
- `ef_filter_spec_set_eth_local()`
- optionally, `ef_filter_spec_set_vlan()`
  
- `ef_filter_spec_set_ip4_local()` or `ef_filter_spec_set_ip6_local()`
- `ef_filter_spec_set_eth_type()`
- `ef_filter_spec_set_eth_local()`
- optionally, `ef_filter_spec_set_vlan()`
  
- `ef_filter_spec_set_eth_local()`
- optionally, `ef_filter_spec_set_vlan()`
  
- `ef_filter_spec_set_unicast_mismatch()`
- optionally, `ef_filter_spec_set_vlan()`
  
- `ef_filter_spec_set_multicast_mismatch()`
- optionally, `ef_filter_spec_set_vlan()`

When the Solarflare adapter is configured to use the low\_latency firmware variant, the following combinations of `ef_filter_spec_set_*`() calls are supported, and are applied in this order:

- `ef_filter_spec_set_ip4_full()` or `ef_filter_spec_set_ip6_full()`
- `ef_filter_spec_set_eth_type()`
  
- `ef_filter_spec_set_ip4_local()` or `ef_filter_spec_set_ip6_local()`

- [ef\\_filter\\_spec\\_set\\_eth\\_type\(\)](#)
  
- [ef\\_filter\\_spec\\_set\\_eth\\_local\(\)](#)
- optionally, [ef\\_filter\\_spec\\_set\\_vlan\(\)](#)
  
- [ef\\_filter\\_spec\\_set\\_unicast\\_mismatch\(\)](#)
  
- [ef\\_filter\\_spec\\_set\\_multicast\\_mismatch\(\)](#)

When the Solarflare adapter is configured to use the `packed_stream` firmware variant, the following combinations of `ef_filter_spec_set_*`() calls are supported, and are applied in this order:

- [ef\\_filter\\_spec\\_set\\_ip4\\_local\(\)](#) or [ef\\_filter\\_spec\\_set\\_ip6\\_local\(\)](#)
- [ef\\_filter\\_spec\\_set\\_eth\\_type\(\)](#)
  
- [ef\\_filter\\_spec\\_set\\_eth\\_local\(\)](#)
  
- [ef\\_filter\\_spec\\_set\\_unicast\\_mismatch\(\)](#)
- optionally, [ef\\_filter\\_spec\\_set\\_vlan\(\)](#)
  
- [ef\\_filter\\_spec\\_set\\_multicast\\_mismatch\(\)](#)
- optionally, [ef\\_filter\\_spec\\_set\\_vlan\(\)](#)

## 6.9 IGMP subscriptions

In theory it is possible to generate IGMP join/leave messages and send them via `ef_vi`. For reliable operation this would mean re-implementing the whole IGMP protocol. As the IGMP messages are not especially latency sensitive, this would be a lot of work for minimal benefit.

A common solution when using `ef_vi` to receive multicast is to also create a standard UDP socket and use `IP_ADD_MEMBERSHIP/IP_DROP_MEMBERSHIP` to control the multicast groups for the interface. The kernel stack will then generate and respond to the IGMP messages as normal. This UDP socket should not be accelerated using Onload, which can be achieved by using `onload_socket_nonaccel()` to create this socket.

## Note

The `onload_socket_nonaccel()` function is a part of the Onload Extensions API, and is documented in the *Onload User Guide*.

If the `ef_vi` application installs filters for individual UDP ports then everything will work fine. Once the VI has a UDP filter installed for the multicast traffic, this filter will match the received traffic in preference to the MAC multicast address filter installed by the kernel stack. Consequently, the UDP socket won't see the actual traffic, but the VI will. The IGMP messages will not match the `ef_vi` filter, and so IGMP will still be handled by the kernel stack. If the traffic rate is high, then it is best to install the filter on the VI before joining the group, and to close the UDP socket before removing the filter or closing the VI. This prevents the UDP socket receiving lots of unwanted traffic.

If the `ef_vi` application installs the "multicast-all" filter for the VI, then this will normally consume all multicast traffic on the interface. This would potentially include IGMP messages, which could cause the subscriptions to fail. In this case, an extra filter entry can be added to redirect all incoming IGMP packets to the kernel stack, so that they can still be processed by the kernel:

```
ethtool -U <interface> flow-type ip4 l4proto 2 vlan 0 m 0xf000 action 0
```

## 6.10 Freeing Resources

Users of `ef_vi` must do the following to free resources:

1. Release and free memory regions by calling `ef_memreg_free()`.
2. Release and free a virtual interface by calling `ef_vi_free()`.
3. Release and free a protection domain by calling `ef_pd_free()`.
4. Close a driver handle by calling `ef_driver_close()`.

```
/* Release and free a memory region */  
int ef_memreg_free(ef_memreg* mr,  
                  ef_driver_handle mr_dh);
```

### Figure: Release and Free a Memory Region

```
/* Release and free a virtual interface */  
int ef_vi_free(ef_vi* vi,  
              ef_driver_handle vi_dh);
```

### Figure: Release and Free a Virtual Interface

```
/* Release and free a protection domain */  
int ef_pd_free(ef_pd *pd,  
              ef_driver_handle pd_dh);
```

### Figure: Release and Free a Protection Domain

## 6.11 Design Considerations

This section outlines some considerations that are required when designing an ef\_vi application.

### 6.11.1 Interrupts

Interrupts are not enabled by ef\_vi by default.

Interrupts are enabled only if `ef_eventq_wait()` is called. If there are no events immediately ready, then this function will enable an interrupt, and sleep until that interrupt fires. Interrupts are then disabled again.

### 6.11.2 Thread Safety

There is no thread-safety on ef\_vi functions. This is for speed. If thread-safety is required, it must be provided by the ef\_vi application.

The usual use-case is to have multiple virtual interface structures for independent operation. There is then no need to lock.

### 6.11.3 Packet Buffer Addressing

Different configuration modes are available for addressing packet buffers.

- *Network Adapter Buffer Table Mode* uses a block of memory on the adapter to store a translation table mapping between buffer IDs and physical addresses:
  - When using a SFN5000 or SFN6000 series adapter there are 65536 entries in the buffer table. Each entry maps a 4KB page of memory that holds two 2KB packet buffers, and so a maximum of 131072 packet buffers are available. The kernel uses some of these, leaving about 120,000 packet buffers available for ef\_vi.
  - The SFN7000 series adapters have *Large Buffer Table Support*. Each entry can map a larger region of memory, or a huge page, enabling them to support many more packet buffers without the need to use Scalable Packet Buffer Mode.

This is the default mode.

- *Scalable Packet Buffer Mode* allocates packet buffers from the kernel IOMMU. It uses Single Root I/O Virtualization (SR-IOV) virtual functions (VF) to provide memory protection and translation. This removes the buffer limitation of the buffer table.
  - SR-IOV must be enabled
  - the kernel must support an IOMMU.

An ef\_vi application can enable this mode by setting the environment variable `EF_VI_PD_FLAGS=vf`.

- *Physical Addressing Mode* uses actual physical addresses to identify packet buffers. An ef\_vi application can therefore direct the adapter to access memory that is not in the application address space. For example, this can be used for zero-copy from the kernel buffer cache. any piece of memory.

Physical Addressing Mode allows stacks to use large amounts of packet buffer memory, avoiding address translation limitations on some adapters and without the need to configure and use SR-IOV virtual functions.

- No memory protection is provided. Physical addressing mode removes memory protection from the network adapter's access of packet buffers. Unprivileged user-level code is provided and directly handles the raw physical memory addresses of packet buffers. User-level code provides physical memory addresses directly to the adapter with the ability to read or write arbitrary memory locations.
- It is important to ensure that packet buffers are page aligned.

An ef\_vi application can enable this mode by setting the environment variable `EF_VI_PD_FLAGS=phys`.

The `sfc_char` module option must also be enabled in a file in the `/etc/modprobe.d` directory where `N` is the integer group ID of the user running the ef\_vi application. Set to `-1` means ALL users are permitted access.:

```
options sfc_char phys_mode_gid=<N>
```

For more information about these configuration modes, see the chapter titled *Packet Buffers* in the *Onload User Guide* (SF-104474-CD).

## 6.11.4 Virtual machines

Ef\_vi can be used in virtual machines provided PCI passthrough is used. With PCI passthrough a slice of the network adapter is mapped directly into the address space of the virtual machine so that device drivers in the VM OS can access the network adapter directly. To isolate VMs from one another and from the hypervisor, I/O addresses are also virtualised. I/O addresses generated by the network adapter are translated to physical memory addresses by the system IOMMU.

When ef\_vi is used in virtual machines two levels of address translation are performed by default. Firstly a translation from DMA address to I/O address, performed by the adapter to isolate the application from other applications and the kernel. Then a translation from I/O address to physical address by the system IOMMU, isolating the virtual machines. Physical address mode can also be used in virtual machines, in which case the adapter translation is omitted.

## 6.12 Known Limitations

### 6.12.1 Timestamping

When timestamping is enabled, the VI must be polled regularly (even when no packets are available). Timestamps consume four event queue slots per second and failing to poll the VI can result in event queue overflow. When there are no packets to receive, stale timestamps are not returned to the application, but polling ensures that they are cleared from the event queue.

### 6.12.2 Minimum Fill Level

You must poll for at least `EF_VI_EVENT_POLL_MIN_EVS` at a time. You will also need to initially fill the RX ring with at least 16 packet buffers to ensure that the card begins acquiring packets. (It's OK to underrun once the application has started; although of course doing so risks drops.)

It's (very slightly) more efficient to refill the ring in batch sizes of 8/16/32 or 64 anyway.

## 6.13 Example

Below is a simple example showing a starting framework for an ef\_vi application. This is not a complete program. There is no initialization, and much of the other required code is only indicated by comments.

```
static void handle_poll(ef_vi *vi)
{
    ef_event events[POLL_BATCH_SIZE];
    int n_ev = ef_eventq_poll(&vi, events, POLL_BATCH_SIZE);
    for( i = 0; i < n_ev; ++i ) {
        switch( EF_EVENT_TYPE(events[i]) ) {
            case EF_EVENT_TYPE_RX:
                // Accumulate used buffer
                break;
            case EF_EVENT_TYPE_TX:
                /* Each EF_EVENT_TYPE_TX can signal multiple completed sends */
                int num_completed = ef_vi_transmit_unbundle(vi, events[i], &dma_id);
                break;
            case EF_EVENT_TYPE_RX_DISCARD:
            case EF_EVENT_TYPE_RX_NO_DESC_TRUNC:
                /* Discard events also use up buffers */
                // Accumulate buffer in user space
                break;
            default:
                /* Other error types */
        }
    }
}

static void refill_rx_ring(ef_vi *vi)
{
    if( ef_vi_receive_space(&vi) < REFILL_BATCH_SIZE )
        return;
    int refill_count = REFILL_BATCH_SIZE;
    /* Falling too low? */
    if( ef_vi_receive_space(&vi) > ef_vi_receive_capacity(&vi) / 2 )
        refill_count = ef_vi_receive_space(&vi);
    /* Enough free buffers? */
    if( refill_count > free_bufs_in_sw )
        refill_count = free_bufs_in_sw;
    /* Round down to batch size */
    refill_count &= ~(REFILL_BATCH_SIZE - 1);
    if( refill_count ) {
        while( refill_count ) {
            ef_vi_receive_init(...);
            --refill_count;
        }
        ef_vi_receive_push(&vi);
    }
}

int main(int argc, char argv[]) {
    while( 1 ) {
        poll_events(&vi);
        refill_rx_ring(&vi);
    }
}
```



## Chapter 7

# Worked Example

This part of the documentation examines a simplified version of [eflatency](#). This is a small application which listens for packets and replies, with as low latency as possible.

This documentation discusses the tradeoffs that have been chosen, some performance issues to avoid, and some possible additions that have been omitted for clarity.

See also the supplied code for [eflatency](#), which includes many of these improvements.

### 7.1 Setup

The first step is to set up `ef_vi`:

- #include the various headers we need ([etherfabric/pd.h](#), [vi.h](#), [memreg.h](#))
- open the driver
- allocate a protection domain
- allocate a virtual interface from the protection domain.

```
ef_driver_handle driver_handle;
ef_vi            vi;
ef_pd           pd;
static void do_init(int ifindex){
    ef_driver_open(&driver_handle);
    ef_pd_alloc(&pd, driver_handle, ifindex, EF_PD_DEFAULT );
    ef_vi_alloc_from_pd(&vi, driver_handle, &pd, driver_handle,
                      -1, -1, -1, NULL, -1, 0);
}
```

The following improvements could be made:

- check the return values from these functions, in case the card has run out of resources and is unable to allocate more virtual interfaces
- offer physical buffer mode here.

## 7.2 Creating Packet buffers

The next step is to allocate a memory region and register it for packet buffers.

```
const int BUF_SIZE = 2048; /* Hardware always wants 2k buffers */
int bytes = N_BUFS * BUF_SIZE;
void* p;
posix_memalign(&p, 4096, bytes) /* allocate aligned memory */
ef_memreg_alloc(&memreg, driver_handle, &pd, driver_handle,
                p, bytes); /* Make it available to ef_vi */
```

This is all that is strictly necessary to set up the packet buffers.

However, the packet buffer is 2048 bytes long, whereas the normal MTU size for a transmitted packet is only 1500 bytes. There is some spare memory in each packet buffer. Performance can be improved by using this space to cache some of the packet meta-data, so that it does not have to be recalculated:

- The DMA address of the packet is cached. It is determined by getting the DMA address of the base of the memory chunk, and then incrementing in 2KB chunks.
- The packet ID is also cached.

A structure is used to store the cached meta-data and a few pointers in the buffer. An array is used to track all the buffers:

```
#define MEMBER_OFFSET(c_type, mbr_name) \
    ((uint32_t) (uintptr_t) (&((c_type*)0)->mbr_name))
#define CACHE_ALIGN __attribute__((aligned(EF_VI_DMA_ALIGN)))
struct pkt_buf {
    struct pkt_buf* next;
    /* We're not actually going to use this;
       but chaining multiple buffers together is a common and useful trick. */
    ef_addr dma_buf_addr;
    int id;
    uint8_t dma_buf[1] CACHE_ALIGN;
    /* Not strictly required, but cache aligning the payload is a speed
       boost, so do it. */
};
/* We're also going to want to keep track of all our buffers, so have an
   array of them. Not strictly needed, but convenient. */
struct pkt_buf* pkt_bufs [N_BUFS];
for( i = 0; i < N_BUFS; ++i ) {
    struct pkt_buf* pb = (struct pkt_buf*) ((char*) p + i * 2048);
    pb->id = i;
    pb->dma_buf_addr = ef_memreg_dma_addr(&memreg, i * 2048);
    pb->dma_buf_addr += MEMBER_OFFSET(struct pkt_buf, dma_buf);
    pkt_bufs[i] = pb;
}
```

### Note

When receiving, the hardware will only fill up to 1824 bytes per buffer. Larger jumbo frames are split across multiple buffers.

## 7.3 Adding Filters

Next, a filter is specified and added, so that the virtual interface receives traffic. Assuming there is a sockaddr to work from:

```
struct sockaddr_in sa_local; /* TODO: Fill this out somehow */
ef_filter_spec filter_spec;
ef_filter_spec_init(&filter_spec, EF_FILTER_FLAG_NONE);
TRY(ef_filter_spec_set_ip4_local(&filter_spec, IPPROTO_UDP,
                                sa_local.sin_addr.s_addr,
                                sa_local.sin_port));
TRY(ef_vi_filter_add(&vi, driver_handle, &filter_spec, NULL));
```

## 7.4 Receiving packets

At this point, packets will start arriving at the interface, be diverted to the application, and immediately be dropped.

So the next step is to push some packet buffers to the RX descriptor ring, to receive the incoming packets.

For efficiency, the code pushes packet buffers eight at a time.

```
unsigned rx_posted = 0; /* We need to keep track of which buffers are
                        already on the ring */
void rx_post( int n ) {
    for( int i = 0; i < n; ++i ) {
        struct pkt_buf* pb = pkt_bufs[rx_posted % N_RX_BUFS];
        ef_vi_receive_init(&vi, pb->dma_buf_addr, pb->id);
        ++rx_posted;
    }
    ef_vi_receive_push(&vi);
}
```

So now, there are packet buffers on the descriptor ring. But once they are filled, the application will start dropping again.

## 7.5 Handling Events

The next step is to handle these incoming packets, by polling the event queue.

```
void rx_wait(void) {
    /* Again, for efficiency, poll multiple events at once. */
    ef_event evs[ NUM_POLL_EVENTS ];
    int n_ev, i;

    while( 1 ) {
        n_ev = ef_eventq_poll(&vi, evs, NUM_POLL_EVENTS);
        if( n_ev > 0 )
            for( i = 0; i < n_ev; ++i )
                switch( EF_EVENT_TYPE(evs[i]) ) {
                    case EF_EVENT_TYPE_RX:
                        handle_rx_packet(EF_EVENT_RX_RQ_ID(evs[i]),
                                         EF_EVENT_RX_BYTES(evs[i]) );
                        break;
                    case EF_EVENT_TYPE_RX_DISCARD:
                        /* Interesting to print out the cause of the discard */
                        fprintf(stderr, "ERROR: RX_DISCARD type=%d",
                                EF_EVENT_RX_DISCARD_TYPE(evs[i]));
                        /* but let's handle it like a normal packet anyway */
                        handle_rx_packet(EF_EVENT_RX_RQ_ID(evs[i]),
                                         EF_EVENT_RX_BYTES(evs[i]) );
                        break;
                }
    }
}
```

This code is calling a `handle_rx_packet()` function, passing it the packet id (which corresponds directly to the `pkt_bufs` array - see [Creating Packet buffers](#)) and the length of data. The body of this function is not shown, but it should do the following:

- note that this packet buffer has been consumed, and so is ready to be re-posted:
  - the `rx_post()` function (see [Receiving packets](#)) must also be updated to use this information, so a buffer is not re-posted until it is marked as consumed
- ensure that the received packet is processed according to the purpose of the application:
  - if the application can always process incoming packets as fast as they are received, then it can do its work inline, and immediately repost the buffer on the ring
  - otherwise, the application should probably post an item on a work queue for another thread to act upon, and arrange for the refill to come from a larger pool of buffers
- optionally, handle discards in some different way (perhaps not raising the work event).

## 7.6 Transmitting packets

The next step is to implement the transmit side. The hard part is filling out the payload, and getting all the fields of IP and UDP correct. (ef\_vi is usually used to transmit UDP, as it's a much simpler protocol to implement than TCP.)

There's some sample code to fill out the headers in the functions `ci*_hdr_init()`, which can be found in `src/lib/citools/ippacket.c`.

After that, to transmit one of the `pb` structures (see [Creating Packet buffers](#)), a single function call is needed:

```
ef_vi_transmit(&vi, pb->dma_buf_addr, frame_length, 0);
```

But the application must also keep track of when that buffer is used, and when it is free. This means adding some complexity to the poll loop (see [Handling Events](#)). The absolute minimum is:

```
case EF_EVENT_TYPE_TX:
    ef_vi_transmit_unbundle(&vi, &evs[i], ids);
    break;
```

This is only sufficient if the TX buffers and the RX buffers are from different pools.

### Note

In ping pong there is only ever one outstanding send. The application does not transmit another packet until the remote side has processed the current one, and so the application does not even need to keep track of its state.

One option would be to free up sent buffers, to a pool ready to be filled with data. Other applications may instead fill a few packet buffers with data, and then transmit them as and when, only keeping track to make sure that the TX ring never overfills.

## Chapter 8

# Data Structure Index

### 8.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">ef_event</a>	A token that identifies something that has happened . . . . .	51
<a href="#">ef_eventq_state</a>	State of event queue . . . . .	55
<a href="#">ef_filter_cookie</a>	Cookie identifying a filter . . . . .	57
<a href="#">ef_filter_spec</a>	Specification of a filter . . . . .	58
<a href="#">ef_iovec</a>	Ef_iovec is similar to the standard struct iovec. An array of these is used to designate a scatter/gather list of I/O buffers . . . . .	59
<a href="#">ef_memreg</a>	Memory that has been registered for use with <a href="#">ef_vi</a> . . . . .	60
<a href="#">ef_packed_stream_packet</a>	Per-packet meta-data . . . . .	61
<a href="#">ef_packed_stream_params</a>	Packed-stream mode parameters . . . . .	63
<a href="#">ef_pd</a>	A protection domain . . . . .	65
<a href="#">ef_pio</a>	A Programmed I/O region . . . . .	67
<a href="#">ef_vi</a>	A virtual interface . . . . .	68
<a href="#">ef_vi_layout_entry</a>	Layout of the data that is delivered into receive buffers . . . . .	78
<a href="#">ef_vi_nic_type</a>	The type of NIC in use . . . . .	79
<a href="#">ef_vi_rxq</a>	RX descriptor ring . . . . .	80
<a href="#">ef_vi_rxq_state</a>	State of RX descriptor ring . . . . .	82
<a href="#">ef_vi_set</a>	A virtual interface set within a protection domain . . . . .	84

<a href="#">ef_vi_state</a>	State of a virtual interface . . . . .	85
<a href="#">ef_vi_stats</a>	Statistics for a virtual interface . . . . .	86
<a href="#">ef_vi_stats_field_layout</a>	Layout for a field of statistics . . . . .	87
<a href="#">ef_vi_stats_layout</a>	Layout for statistics . . . . .	88
<a href="#">ef_vi_transmit_alt_overhead</a>	Per-packet overhead information . . . . .	90
<a href="#">ef_vi_txq</a>	TX descriptor ring . . . . .	91
<a href="#">ef_vi_txq_state</a>	State of TX descriptor ring . . . . .	92
<a href="#">ef_vi::ops</a>	Driver-dependent operations . . . . .	93

## Chapter 9

# File Index

### 9.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">base.h</a>	Base definitions for EtherFabric Virtual Interface HAL . . . . .	104
<a href="#">capabilities.h</a>	Capabilities API for EtherFabric Virtual Interface HAL . . . . .	107
<a href="#">checksum.h</a>	Checksum utility functions . . . . .	111
<a href="#">ef_vi.h</a>	Virtual Interface definitions for EtherFabric Virtual Interface HAL . . . . .	115
<a href="#">memreg.h</a>	Registering memory for EtherFabric Virtual Interface HAL . . . . .	155
<a href="#">packedstream.h</a>	Packed streams for EtherFabric Virtual Interface HAL . . . . .	158
<a href="#">pd.h</a>	Protection Domains for EtherFabric Virtual Interface HAL . . . . .	162
<a href="#">pio.h</a>	Programmed Input/Output for EtherFabric Virtual Interface HAL . . . . .	166
<a href="#">timer.h</a>	Timers for EtherFabric Virtual Interface HAL . . . . .	171
<a href="#">vi.h</a>	Virtual packet / DMA interface for EtherFabric Virtual Interface HAL . . . . .	175





## Chapter 10

# Data Structure Documentation

### 10.1 ef\_event Union Reference

A token that identifies something that has happened.

```
#include <ef_vi.h>
```

#### Data Fields

- struct {  
    unsigned **type**:16  
} **generic**
- struct {  
    unsigned **type**:16  
    unsigned **q\_id**:8  
    unsigned **\_\_reserved**:8  
    unsigned **rq\_id**:32  
    unsigned **len**:16  
    unsigned **flags**:16  
} **rx**
- struct {  
    unsigned **type**:16  
    unsigned **q\_id**:8  
    unsigned **\_\_reserved**:8  
    unsigned **rq\_id**:32  
    unsigned **len**:16  
    unsigned **flags**:16  
    unsigned **subtype**:16  
} **rx\_discard**
- struct {  
    unsigned **type**:16  
    unsigned **q\_id**:8  
    unsigned **flags**:8  
    unsigned **desc\_id**:16  
} **tx**

- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
  - unsigned **flags**:8
  - unsigned **desc\_id**:16
  - unsigned **subtype**:16
 } [tx\\_error](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
  - unsigned **flags**:8
  - unsigned **rq\_id**:32
  - unsigned **ts\_sec**:32
  - unsigned **ts\_nsec**:32
 } [tx\\_timestamp](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
  - unsigned **\_\_reserved**:8
  - unsigned **alt\_id**:16
 } [tx\\_alt](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
 } [rx\\_no\\_desc\\_trunc](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
  - unsigned **\_\_reserved**:8
  - unsigned **flags**:16
  - unsigned **n\_pkts**:16
  - unsigned **ps\_flags**:8
 } [rx\\_packed\\_stream](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **data**
 } [sw](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
  - unsigned **\_\_reserved**:8
  - unsigned **n\_descs**:16
  - unsigned **flags**:16
 } [rx\\_multi](#)
  
- struct {
  - unsigned **type**:16
  - unsigned **q\_id**:8
  - unsigned **\_\_reserved**:8
  - unsigned **n\_descs**:16

```
unsigned flags:16  
unsigned subtype:16  
} rx_multi_discard
```

### 10.1.1 Detailed Description

A token that identifies something that has happened.

Examples include packets received, packets transmitted, and errors.

Users should not access this structure, but should instead use the macros provided.

Definition at line 165 of file ef\_vi.h.

### 10.1.2 Field Documentation

#### 10.1.2.1 generic

```
struct { ... } generic
```

A generic event, to query the type when it is unknown

#### 10.1.2.2 rx

```
struct { ... } rx
```

An event of type EF\_EVENT\_TYPE\_RX

#### 10.1.2.3 rx\_discard

```
struct { ... } rx_discard
```

An event of type EF\_EVENT\_TYPE\_RX\_DISCARD

#### 10.1.2.4 rx\_multi

```
struct { ... } rx_multi
```

An event of type EF\_EVENT\_TYPE\_RX\_MULTI

#### 10.1.2.5 rx\_multi\_discard

```
struct { ... } rx_multi_discard
```

An event of type EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD

#### 10.1.2.6 rx\_no\_desc\_trunc

```
struct { ... } rx_no_desc_trunc
```

An event of type EF\_EVENT\_TYPE\_RX\_NO\_DESC\_TRUNC

#### 10.1.2.7 rx\_packed\_stream

```
struct { ... } rx_packed_stream
```

An event of type EF\_EVENT\_TYPE\_RX\_PACKED\_STREAM

#### 10.1.2.8 sw

```
struct { ... } sw
```

An event of type EF\_EVENT\_TYPE\_SW

#### 10.1.2.9 tx

```
struct { ... } tx
```

An event of type EF\_EVENT\_TYPE\_TX

#### 10.1.2.10 tx\_alt

```
struct { ... } tx_alt
```

An event of type EF\_EVENT\_TYPE\_TX\_ALT

#### 10.1.2.11 tx\_error

```
struct { ... } tx_error
```

An event of type EF\_EVENT\_TYPE\_TX\_ERROR

### 10.1.2.12 tx\_timestamp

```
struct { ... } tx_timestamp
```

An event of type EF\_EVENT\_TYPE\_TX\_WITH\_TIMESTAMP

The documentation for this union was generated from the following file:

- [ef\\_vi.h](#)

## 10.2 ef\_eventq\_state Struct Reference

State of event queue.

```
#include <ef_vi.h>
```

### Data Fields

- [ef\\_eventq\\_ptr evq\\_ptr](#)
- [int32\\_t evq\\_clear\\_stride](#)
- [uint32\\_t sync\\_timestamp\\_major](#)
- [uint32\\_t sync\\_timestamp\\_minor](#)
- [uint32\\_t sync\\_timestamp\\_minimum](#)
- [uint32\\_t sync\\_timestamp\\_synchronised](#)
- [uint32\\_t sync\\_flags](#)

### 10.2.1 Detailed Description

State of event queue.

Users should not access this structure.

Definition at line 625 of file ef\_vi.h.

### 10.2.2 Field Documentation

#### 10.2.2.1 evq\_clear\_stride

```
int32_t evq_clear_stride
```

For internal use only

Definition at line 629 of file ef\_vi.h.

### 10.2.2.2 `evq_ptr`

`ef_eventq_ptr` `evq_ptr`

Event queue pointer

Definition at line 627 of file `ef_vi.h`.

### 10.2.2.3 `sync_flags`

`uint32_t` `sync_flags`

Time synchronization flags

Definition at line 639 of file `ef_vi.h`.

### 10.2.2.4 `sync_timestamp_major`

`uint32_t` `sync_timestamp_major`

Timestamp (major part)

Definition at line 631 of file `ef_vi.h`.

### 10.2.2.5 `sync_timestamp_minimum`

`uint32_t` `sync_timestamp_minimum`

Smallest possible seconds value for given `sync_timestamp_major`

Definition at line 635 of file `ef_vi.h`.

### 10.2.2.6 `sync_timestamp_minor`

`uint32_t` `sync_timestamp_minor`

Timestamp (minor part)

Definition at line 633 of file `ef_vi.h`.

### 10.2.2.7 sync\_timestamp\_synchronised

```
uint32_t sync_timestamp_synchronised
```

Timestamp synchronized with adapter

Definition at line 637 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.3 ef\_filter\_cookie Struct Reference

Cookie identifying a filter.

```
#include <vi.h>
```

### Data Fields

- int [filter\\_id](#)
- int [filter\\_type](#)

### 10.3.1 Detailed Description

Cookie identifying a filter.

Definition at line 485 of file vi.h.

### 10.3.2 Field Documentation

#### 10.3.2.1 filter\_id

```
int filter_id
```

ID of the filter

Definition at line 487 of file vi.h.

### 10.3.2.2 filter\_type

```
int filter_type
```

Type of the filter

Definition at line 489 of file vi.h.

The documentation for this struct was generated from the following file:

- [vi.h](#)

## 10.4 ef\_filter\_spec Struct Reference

Specification of a filter.

```
#include <vi.h>
```

### Data Fields

- unsigned [type](#)
- unsigned [flags](#)
- unsigned [data](#) [12]

### 10.4.1 Detailed Description

Specification of a filter.

Definition at line 469 of file vi.h.

### 10.4.2 Field Documentation

#### 10.4.2.1 data

```
unsigned data[12]
```

Data for filter

Definition at line 475 of file vi.h.



### 10.4.2.2 flags

unsigned flags

Flags for filter

Definition at line 473 of file vi.h.

### 10.4.2.3 type

unsigned type

Type of filter

Definition at line 471 of file vi.h.

The documentation for this struct was generated from the following file:

- [vi.h](#)

## 10.5 ef\_iovec Struct Reference

[ef\\_iovec](#) is similar to the standard struct `iovec`. An array of these is used to designate a scatter/gather list of I/O buffers.

```
#include <ef_vi.h>
```

### Public Member Functions

- [ef\\_addr](#) iov\_base [EF\\_VI\\_ALIGN](#) (8)

### Data Fields

- unsigned [iov\\_len](#)

#### 10.5.1 Detailed Description

[ef\\_iovec](#) is similar to the standard struct `iovec`. An array of these is used to designate a scatter/gather list of I/O buffers.

Definition at line 448 of file `ef_vi.h`.

## 10.5.2 Member Function Documentation

### 10.5.2.1 EF\_VI\_ALIGN()

```
ef_addr iov_base EF_VI_ALIGN (  
    8 )
```

base address of the buffer

## 10.5.3 Field Documentation

### 10.5.3.1 iov\_len

```
unsigned iov_len
```

length of the buffer

Definition at line 452 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.6 ef\_memreg Struct Reference

Memory that has been registered for use with [ef\\_vi](#).

```
#include <memreg.h>
```

### Data Fields

- [ef\\_addr](#) \* [mr\\_dma\\_addrs](#)
- [ef\\_addr](#) \* [mr\\_dma\\_addrs\\_base](#)

### 10.6.1 Detailed Description

Memory that has been registered for use with [ef\\_vi](#).

Definition at line 48 of file memreg.h.

## 10.6.2 Field Documentation

### 10.6.2.1 mr\_dma\_addrs

`ef_addr*` `mr_dma_addrs`

Addresses of DMA buffers within the reserved system memory

Definition at line 50 of file `memreg.h`.

### 10.6.2.2 mr\_dma\_addrs\_base

`ef_addr*` `mr_dma_addrs_base`

Base addresses of reserved system memory

Definition at line 52 of file `memreg.h`.

The documentation for this struct was generated from the following file:

- [memreg.h](#)

## 10.7 ef\_packed\_stream\_packet Struct Reference

Per-packet meta-data.

```
#include <packedstream.h>
```

### Data Fields

- `uint16_t` [ps\\_next\\_offset](#)
- `uint8_t` [ps\\_pkt\\_start\\_offset](#)
- `uint8_t` [ps\\_flags](#)
- `uint16_t` [ps\\_cap\\_len](#)
- `uint16_t` [ps\\_orig\\_len](#)
- `uint32_t` [ps\\_ts\\_sec](#)
- `uint32_t` [ps\\_ts\\_nsec](#)

### 10.7.1 Detailed Description

Per-packet meta-data.

Definition at line 49 of file `packedstream.h`.

## 10.7.2 Field Documentation

### 10.7.2.1 ps\_cap\_len

`uint16_t ps_cap_len`

Number of bytes of packet payload stored.

Definition at line 57 of file packedstream.h.

### 10.7.2.2 ps\_flags

`uint8_t ps_flags`

EF\_VI\_PS\_FLAG\_\* flags.

Definition at line 55 of file packedstream.h.

### 10.7.2.3 ps\_next\_offset

`uint16_t ps_next_offset`

Offset of next packet from start of this struct.

Definition at line 51 of file packedstream.h.

### 10.7.2.4 ps\_orig\_len

`uint16_t ps_orig_len`

Length of the frame on the wire.

Definition at line 59 of file packedstream.h.

### 10.7.2.5 ps\_pkt\_start\_offset

```
uint8_t ps_pkt_start_offset
```

Offset of packet payload from start of this struct.

Definition at line 53 of file packedstream.h.

### 10.7.2.6 ps\_ts\_nsec

```
uint32_t ps_ts_nsec
```

Hardware timestamp (nanoseconds).

Definition at line 63 of file packedstream.h.

### 10.7.2.7 ps\_ts\_sec

```
uint32_t ps_ts_sec
```

Hardware timestamp (seconds).

Definition at line 61 of file packedstream.h.

The documentation for this struct was generated from the following file:

- [packedstream.h](#)

## 10.8 ef\_packed\_stream\_params Struct Reference

Packed-stream mode parameters.

```
#include <packedstream.h>
```

### Data Fields

- int [psp\\_buffer\\_size](#)
- int [psp\\_buffer\\_align](#)
- int [psp\\_start\\_offset](#)
- int [psp\\_max\\_usable\\_buffers](#)

## 10.8.1 Detailed Description

Packed-stream mode parameters.

The application should query these parameters using `ef_vi_packed_stream_get_params()` to determine buffer size etc.

Definition at line 92 of file `packedstream.h`.

## 10.8.2 Field Documentation

### 10.8.2.1 `psp_buffer_align`

```
int psp_buffer_align
```

Alignment requirement for start of packed-stream buffers.

Definition at line 97 of file `packedstream.h`.

### 10.8.2.2 `psp_buffer_size`

```
int psp_buffer_size
```

Size of each packed-stream buffer.

Definition at line 94 of file `packedstream.h`.

### 10.8.2.3 `psp_max_usable_buffers`

```
int psp_max_usable_buffers
```

The maximum number of packed-stream buffers that the adapter can deliver packets into without software consuming any completion events. The application can post more buffers, but they will only be used as events are consumed.

Definition at line 109 of file `packedstream.h`.

#### 10.8.2.4 psp\_start\_offset

```
int psp_start_offset
```

Offset from the start of a packed-stream buffer to where the first packet will be delivered.

Definition at line 102 of file packedstream.h.

The documentation for this struct was generated from the following file:

- [packedstream.h](#)

## 10.9 ef\_pd Struct Reference

A protection domain.

```
#include <pd.h>
```

### Data Fields

- enum [ef\\_pd\\_flags](#) [pd\\_flags](#)
- unsigned [pd\\_resource\\_id](#)
- char \* [pd\\_intf\\_name](#)
- char \* [pd\\_cluster\\_name](#)
- int [pd\\_cluster\\_sock](#)
- [ef\\_driver\\_handle](#) [pd\\_cluster\\_dh](#)
- unsigned [pd\\_cluster\\_viset\\_resource\\_id](#)
- int [pd\\_cluster\\_viset\\_index](#)

### 10.9.1 Detailed Description

A protection domain.

Definition at line 77 of file pd.h.

### 10.9.2 Field Documentation

#### 10.9.2.1 pd\_cluster\_dh

```
ef\_driver\_handle pd_cluster_dh
```

Driver handle for the application cluster associated with the protection domain

Definition at line 93 of file pd.h.

### 10.9.2.2 `pd_cluster_name`

```
char* pd_cluster_name
```

Name of the application cluster associated with the protection domain

Definition at line 87 of file pd.h.

### 10.9.2.3 `pd_cluster_sock`

```
int pd_cluster_sock
```

Socket for the application cluster associated with the protection domain

Definition at line 90 of file pd.h.

### 10.9.2.4 `pd_cluster_viset_index`

```
int pd_cluster_viset_index
```

Index of VI wanted within a cluster.

Definition at line 98 of file pd.h.

### 10.9.2.5 `pd_cluster_viset_resource_id`

```
unsigned pd_cluster_viset_resource_id
```

Resource ID of the virtual interface set for the application cluster associated with the protection domain

Definition at line 96 of file pd.h.

### 10.9.2.6 `pd_flags`

```
enum ef_pd_flags pd_flags
```

Flags for the protection domain

Definition at line 79 of file pd.h.



### 10.9.2.7 pd\_intf\_name

```
char* pd_intf_name
```

Name of the interface associated with the protection domain

Definition at line 83 of file pd.h.

### 10.9.2.8 pd\_resource\_id

```
unsigned pd_resource_id
```

Resource ID of the protection domain

Definition at line 81 of file pd.h.

The documentation for this struct was generated from the following file:

- [pd.h](#)

## 10.10 ef\_pio Struct Reference

A Programmed I/O region.

```
#include <pio.h>
```

### Data Fields

- `uint8_t *` [pio\\_buffer](#)
- `uint8_t *` [pio\\_io](#)
- `unsigned` [pio\\_resource\\_id](#)
- `unsigned` [pio\\_len](#)

### 10.10.1 Detailed Description

A Programmed I/O region.

Definition at line 57 of file pio.h.

### 10.10.2 Field Documentation

### 10.10.2.1 pio\_buffer

```
uint8_t* pio_buffer
```

The buffer for the Programmed I/O region

Definition at line 59 of file pio.h.

### 10.10.2.2 pio\_io

```
uint8_t* pio_io
```

The I/O region of the virtual interface that is linked with the Programmed I/O region

Definition at line 62 of file pio.h.

### 10.10.2.3 pio\_len

```
unsigned pio_len
```

The length of the Programmed I/O region

Definition at line 66 of file pio.h.

### 10.10.2.4 pio\_resource\_id

```
unsigned pio_resource_id
```

The resource ID for the Programmed I/O region

Definition at line 64 of file pio.h.

The documentation for this struct was generated from the following file:

- [pio.h](#)

## 10.11 ef\_vi Struct Reference

A virtual interface.

```
#include <ef_vi.h>
```

## Data Structures

- struct `ops`  
*Driver-dependent operations.*

## Data Fields

- unsigned `inited`
- unsigned `vi_resource_id`
- unsigned `vi_i`
- unsigned `rx_buffer_len`
- unsigned `rx_prefix_len`
- uint64\_t `rx_discard_mask`
- int `rx_ts_correction`
- int `tx_ts_correction_ns`
- enum `ef_timestamp_format` `ts_format`
- char \* `vi_mem_mmap_ptr`
- int `vi_mem_mmap_bytes`
- char \* `vi_io_mmap_ptr`
- int `vi_io_mmap_bytes`
- char \* `vi_ctpio_mmap_ptr`
- uint32\_t `vi_ctpio_wb_ticks`
- int `vi_clustered`
- int `vi_is_packed_stream`
- int `vi_is_normal`
- unsigned `vi_ps_buf_size`
- `ef_vi_ioaddr_t` `io`
- struct `ef_pio` \* `linked_pio`
- char \* `evq_base`
- unsigned `evq_mask`
- unsigned `timer_quantum_ns`
- unsigned `tx_push_thresh`
- `ef_vi_txq` `vi_txq`
- `ef_vi_rxq` `vi_rxq`
- `ef_vi_state` \* `ep_state`
- enum `ef_vi_flags` `vi_flags`
- enum `ef_vi_out_flags` `vi_out_flags`
- `ef_vi_stats` \* `vi_stats`
- struct `ef_vi` \* `vi_qs` [`EF_VI_MAX_QS`]
- int `vi_qs_n`
- unsigned `tx_alt_num`
- unsigned \* `tx_alt_id2hw`
- unsigned \* `tx_alt_hw2id`
- struct `ef_vi_nic_type` `nic_type`
- struct `ef_vi::ops` `ops`

### 10.11.1 Detailed Description

A virtual interface.

An `ef_vi` represents a virtual interface on a specific NIC. A virtual interface is a collection of an event queue and two DMA queues used to pass Ethernet frames between the transport implementation and the network.

Users should not access this structure.

Definition at line 742 of file `ef_vi.h`.

### 10.11.2 Field Documentation

#### 10.11.2.1 `ep_state`

```
ef_vi_state* ep_state
```

The state of the virtual interface

Definition at line 805 of file `ef_vi.h`.

#### 10.11.2.2 `evq_base`

```
char* evq_base
```

Base of the event queue for the virtual interface

Definition at line 790 of file `ef_vi.h`.

#### 10.11.2.3 `evq_mask`

```
unsigned evq_mask
```

Mask for offsets within the event queue for the virtual interface

Definition at line 792 of file `ef_vi.h`.

#### 10.11.2.4 inited

```
unsigned inited
```

True if the virtual interface has been initialized

Definition at line 744 of file ef\_vi.h.

#### 10.11.2.5 io

```
ef_vi_ioaddr_t io
```

I/O address for the virtual interface

Definition at line 784 of file ef\_vi.h.

#### 10.11.2.6 linked\_pio

```
struct ef_pio* linked_pio
```

Programmed I/O region linked to the virtual interface

Definition at line 787 of file ef\_vi.h.

#### 10.11.2.7 nic\_type

```
struct ef_vi_nic_type nic_type
```

The type of NIC hosting the virtual interface

Definition at line 826 of file ef\_vi.h.

#### 10.11.2.8 ops

```
struct ef_vi::ops ops
```

Driver-dependent operations.

#### 10.11.2.9 rx\_buffer\_len

`unsigned rx_buffer_len`

The length of a receive buffer

Definition at line 751 of file ef\_vi.h.

#### 10.11.2.10 rx\_discard\_mask

`uint64_t rx_discard_mask`

The mask to select which errors cause a discard event

Definition at line 755 of file ef\_vi.h.

#### 10.11.2.11 rx\_prefix\_len

`unsigned rx_prefix_len`

The length of the prefix at the start of a received packet

Definition at line 753 of file ef\_vi.h.

#### 10.11.2.12 rx\_ts\_correction

`int rx_ts_correction`

The timestamp correction (ticks) for received packets

Definition at line 757 of file ef\_vi.h.

#### 10.11.2.13 timer\_quantum\_ns

`unsigned timer_quantum_ns`

The timer quantum for the virtual interface, in nanoseconds

Definition at line 794 of file ef\_vi.h.

#### 10.11.2.14 ts\_format

```
enum ef_timestamp_format ts_format
```

The timestamp format used by the hardware

Definition at line 761 of file ef\_vi.h.

#### 10.11.2.15 tx\_alt\_hw2id

```
unsigned* tx_alt_hw2id
```

Mapping from hardware TX alternative IDs to end-user IDs

Definition at line 823 of file ef\_vi.h.

#### 10.11.2.16 tx\_alt\_id2hw

```
unsigned* tx_alt_id2hw
```

Mapping from end-user TX alternative IDs to hardware IDs

Definition at line 821 of file ef\_vi.h.

#### 10.11.2.17 tx\_alt\_num

```
unsigned tx_alt_num
```

Number of TX alternatives for the virtual interface

Definition at line 819 of file ef\_vi.h.

#### 10.11.2.18 tx\_push\_thresh

```
unsigned tx_push_thresh
```

The threshold at which to switch from using TX descriptor push to using a doorbell

Definition at line 798 of file ef\_vi.h.

#### 10.11.2.19 tx\_ts\_correction\_ns

```
int tx_ts_correction_ns
```

The timestamp correction (ns) for transmitted packets

Definition at line 759 of file ef\_vi.h.

#### 10.11.2.20 vi\_clustered

```
int vi_clustered
```

True if the virtual interface is in a cluster

Definition at line 775 of file ef\_vi.h.

#### 10.11.2.21 vi\_ctpio\_mmap\_ptr

```
char* vi_ctpio_mmap_ptr
```

Pointer to CTPIO region

Definition at line 771 of file ef\_vi.h.

#### 10.11.2.22 vi\_ctpio\_wb\_ticks

```
uint32_t vi_ctpio_wb_ticks
```

Controls rate of writes into CTPIO aperture

Definition at line 773 of file ef\_vi.h.

#### 10.11.2.23 vi\_flags

```
enum ef_vi_flags vi_flags
```

The flags for the virtual interface

Definition at line 807 of file ef\_vi.h.



#### 10.11.2.24 vi\_i

```
unsigned vi_i
```

The instance ID of the virtual interface

Definition at line 748 of file ef\_vi.h.

#### 10.11.2.25 vi\_io\_mmap\_bytes

```
int vi_io_mmap_bytes
```

Length of virtual interface I/O region

Definition at line 769 of file ef\_vi.h.

#### 10.11.2.26 vi\_io\_mmap\_ptr

```
char* vi_io_mmap_ptr
```

Pointer to virtual interface I/O region

Definition at line 767 of file ef\_vi.h.

#### 10.11.2.27 vi\_is\_normal

```
int vi_is_normal
```

True if no special mode is enabled for the virtual interface

Definition at line 779 of file ef\_vi.h.

#### 10.11.2.28 vi\_is\_packed\_stream

```
int vi_is_packed_stream
```

True if packed stream mode is enabled for the virtual interface

Definition at line 777 of file ef\_vi.h.

#### 10.11.2.29 vi\_mem\_mmap\_bytes

```
int vi_mem_mmap_bytes
```

Length of virtual interface memory

Definition at line 765 of file ef\_vi.h.

#### 10.11.2.30 vi\_mem\_mmap\_ptr

```
char* vi_mem_mmap_ptr
```

Pointer to virtual interface memory

Definition at line 763 of file ef\_vi.h.

#### 10.11.2.31 vi\_out\_flags

```
enum ef_vi_out_flags vi_out_flags
```

Flags returned when the virtual interface is allocated

Definition at line 809 of file ef\_vi.h.

#### 10.11.2.32 vi\_ps\_buf\_size

```
unsigned vi_ps_buf_size
```

The packed stream buffer size for the virtual interface

Definition at line 781 of file ef\_vi.h.

#### 10.11.2.33 vi\_qs

```
struct ef_vi* vi_qs[EF_VI_MAX_QS]
```

Virtual queues for the virtual interface

Definition at line 814 of file ef\_vi.h.

#### 10.11.2.34 vi\_qs\_n

```
int vi_qs_n
```

Number of virtual queues for the virtual interface

Definition at line 816 of file ef\_vi.h.

#### 10.11.2.35 vi\_resource\_id

```
unsigned vi_resource_id
```

The resource ID of the virtual interface

Definition at line 746 of file ef\_vi.h.

#### 10.11.2.36 vi\_rxq

```
ef_vi_rxq vi_rxq
```

The RX descriptor ring for the virtual interface

Definition at line 803 of file ef\_vi.h.

#### 10.11.2.37 vi\_stats

```
ef_vi_stats* vi_stats
```

Statistics for the virtual interface

Definition at line 811 of file ef\_vi.h.

#### 10.11.2.38 vi\_txq

```
ef_vi_txq vi_txq
```

The TX descriptor ring for the virtual interface

Definition at line 801 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.12 ef\_vi\_layout\_entry Struct Reference

Layout of the data that is delivered into receive buffers.

```
#include <ef_vi.h>
```

### Data Fields

- enum [ef\\_vi\\_layout\\_type](#) `evle_type`
- int [evle\\_offset](#)
- const char \* [evle\\_description](#)

### 10.12.1 Detailed Description

Layout of the data that is delivered into receive buffers.

Definition at line 2148 of file `ef_vi.h`.

### 10.12.2 Field Documentation

#### 10.12.2.1 `evle_description`

```
const char* evle_description
```

Description of the layout

Definition at line 2154 of file `ef_vi.h`.

#### 10.12.2.2 `evle_offset`

```
int evle_offset
```

Offset to the data

Definition at line 2152 of file `ef_vi.h`.

### 10.12.2.3 evle\_type

```
enum ef_vi_layout_type evle_type
```

The type of layout

Definition at line 2150 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.13 ef\_vi\_nic\_type Struct Reference

The type of NIC in use.

```
#include <ef_vi.h>
```

### Data Fields

- unsigned char [arch](#)
- char [variant](#)
- unsigned char [revision](#)
- unsigned char [nic\\_flags](#)

### 10.13.1 Detailed Description

The type of NIC in use.

Users should not access this structure.

Definition at line 701 of file ef\_vi.h.

### 10.13.2 Field Documentation

#### 10.13.2.1 arch

```
unsigned char arch
```

Architecture of the NIC

Definition at line 703 of file ef\_vi.h.

### 10.13.2.2 nic\_flags

```
unsigned char nic_flags
```

Flags indicating hardware features

Definition at line 709 of file ef\_vi.h.

### 10.13.2.3 revision

```
unsigned char revision
```

Revision of the NIC

Definition at line 707 of file ef\_vi.h.

### 10.13.2.4 variant

```
char variant
```

Variant of the NIC

Definition at line 705 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.14 ef\_vi\_rxq Struct Reference

RX descriptor ring.

```
#include <ef_vi.h>
```

### Data Fields

- [uint32\\_t mask](#)
- [void \\* descriptors](#)
- [uint32\\_t \\* ids](#)

## 10.14.1 Detailed Description

RX descriptor ring.

Users should not access this structure.

Definition at line 659 of file ef\_vi.h.

## 10.14.2 Field Documentation

### 10.14.2.1 descriptors

```
void* descriptors
```

Pointer to descriptors

Definition at line 663 of file ef\_vi.h.

### 10.14.2.2 ids

```
uint32_t* ids
```

Pointer to IDs

Definition at line 665 of file ef\_vi.h.

### 10.14.2.3 mask

```
uint32_t mask
```

Mask for indexes within ring, to wrap around

Definition at line 661 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.15 ef\_vi\_rxq\_state Struct Reference

State of RX descriptor ring.

```
#include <ef_vi.h>
```

### Data Fields

- uint32\_t [posted](#)
- uint32\_t [added](#)
- uint32\_t [removed](#)
- uint32\_t [in\\_jumbo](#)
- uint32\_t [bytes\\_acc](#)
- uint16\_t [last\\_desc\\_i](#)
- uint16\_t [rx\\_ps\\_credit\\_avail](#)

### 10.15.1 Detailed Description

State of RX descriptor ring.

Users should not access this structure.

Definition at line 604 of file ef\_vi.h.

### 10.15.2 Field Documentation

#### 10.15.2.1 added

```
uint32_t added
```

Descriptors added to the ring

Definition at line 608 of file ef\_vi.h.

#### 10.15.2.2 bytes\_acc

```
uint32_t bytes_acc
```

Bytes received as part of a jumbo (7000-series only)

Definition at line 614 of file ef\_vi.h.



### 10.15.2.3 in\_jumbo

`uint32_t in_jumbo`

Packets received as part of a jumbo (7000-series only)

Definition at line 612 of file ef\_vi.h.

### 10.15.2.4 last\_desc\_i

`uint16_t last_desc_i`

Last descriptor index completed (7000-series only)

Definition at line 616 of file ef\_vi.h.

### 10.15.2.5 posted

`uint32_t posted`

Descriptors posted to the nic

Definition at line 606 of file ef\_vi.h.

### 10.15.2.6 removed

`uint32_t removed`

Descriptors removed from the ring

Definition at line 610 of file ef\_vi.h.

### 10.15.2.7 rx\_ps\_credit\_avail

`uint16_t rx_ps_credit_avail`

Credit for packed stream handling (7000-series only)

Definition at line 618 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.16 ef\_vi\_set Struct Reference

A virtual interface set within a protection domain.

```
#include <vi.h>
```

### Data Fields

- unsigned [vis\\_res\\_id](#)
- struct [ef\\_pd](#) \* [vis\\_pd](#)

### 10.16.1 Detailed Description

A virtual interface set within a protection domain.

Definition at line 328 of file [vi.h](#).

### 10.16.2 Field Documentation

#### 10.16.2.1 vis\_pd

```
struct ef\_pd* vis\_pd
```

Protection domain from which the virtual interface set is allocated

Definition at line 332 of file [vi.h](#).

#### 10.16.2.2 vis\_res\_id

```
unsigned vis\_res\_id
```

Resource ID for the virtual interface set

Definition at line 330 of file [vi.h](#).

The documentation for this struct was generated from the following file:

- [vi.h](#)

## 10.17 ef\_vi\_state Struct Reference

State of a virtual interface.

```
#include <ef_vi.h>
```

### Data Fields

- [ef\\_eventq\\_state](#) evq
- [ef\\_vi\\_txq\\_state](#) txq
- [ef\\_vi\\_rxq\\_state](#) rxq

### 10.17.1 Detailed Description

State of a virtual interface.

Users should not access this structure.

Definition at line 672 of file ef\_vi.h.

### 10.17.2 Field Documentation

#### 10.17.2.1 evq

```
ef_eventq_state evq
```

Event queue state

Definition at line 674 of file ef\_vi.h.

#### 10.17.2.2 rxq

```
ef_vi_rxq_state rxq
```

RX descriptor ring state

Definition at line 678 of file ef\_vi.h.

### 10.17.2.3 txq

`ef_vi_txq_state` txq

TX descriptor ring state

Definition at line 676 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.18 ef\_vi\_stats Struct Reference

Statistics for a virtual interface.

```
#include <ef_vi.h>
```

### Data Fields

- `uint32_t rx_ev_lost`
- `uint32_t rx_ev_bad_desc_i`
- `uint32_t rx_ev_bad_q_label`
- `uint32_t evq_gap`

### 10.18.1 Detailed Description

Statistics for a virtual interface.

Users should not access this structure.

Definition at line 686 of file ef\_vi.h.

### 10.18.2 Field Documentation

#### 10.18.2.1 evq\_gap

`uint32_t` evq\_gap

Gaps in the event queue (empty slot followed by event)

Definition at line 694 of file ef\_vi.h.

### 10.18.2.2 rx\_ev\_bad\_desc\_i

uint32\_t rx\_ev\_bad\_desc\_i

RX events with a bad descriptor

Definition at line 690 of file ef\_vi.h.

### 10.18.2.3 rx\_ev\_bad\_q\_label

uint32\_t rx\_ev\_bad\_q\_label

RX events with a bad queue label

Definition at line 692 of file ef\_vi.h.

### 10.18.2.4 rx\_ev\_lost

uint32\_t rx\_ev\_lost

RX events lost

Definition at line 688 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.19 ef\_vi\_stats\_field\_layout Struct Reference

Layout for a field of statistics.

```
#include <vi.h>
```

### Data Fields

- char \* [evsfl\\_name](#)
- int [evsfl\\_offset](#)
- int [evsfl\\_size](#)

### 10.19.1 Detailed Description

Layout for a field of statistics.

Definition at line 994 of file vi.h.

### 10.19.2 Field Documentation

#### 10.19.2.1 evsfl\_name

```
char* evsfl_name
```

Name of statistics field

Definition at line 996 of file vi.h.

#### 10.19.2.2 evsfl\_offset

```
int evsfl_offset
```

Offset of statistics field, in bytes

Definition at line 998 of file vi.h.

#### 10.19.2.3 evsfl\_size

```
int evsfl_size
```

Size of statistics field, in bytes

Definition at line 1000 of file vi.h.

The documentation for this struct was generated from the following file:

- [vi.h](#)

## 10.20 ef\_vi\_stats\_layout Struct Reference

Layout for statistics.

```
#include <vi.h>
```

## Data Fields

- [int evsl\\_data\\_size](#)
- [int evsl\\_fields\\_num](#)
- [ef\\_vi\\_stats\\_field\\_layout evsl\\_fields \[\]](#)

### 10.20.1 Detailed Description

Layout for statistics.

Definition at line 1004 of file vi.h.

### 10.20.2 Field Documentation

#### 10.20.2.1 evsl\_data\_size

```
int evsl_data_size
```

Size of data for statistics

Definition at line 1006 of file vi.h.

#### 10.20.2.2 evsl\_fields

```
ef_vi_stats_field_layout evsl_fields[]
```

Array of fields of statistics

Definition at line 1010 of file vi.h.

#### 10.20.2.3 evsl\_fields\_num

```
int evsl_fields_num
```

Number of fields of statistics

Definition at line 1008 of file vi.h.

The documentation for this struct was generated from the following file:

- [vi.h](#)

## 10.21 ef\_vi\_transmit\_alt\_overhead Struct Reference

Per-packet overhead information.

```
#include <ef_vi.h>
```

### Data Fields

- uint32\_t [pre\\_round](#)
- uint32\_t [mask](#)
- uint32\_t [post\\_round](#)

### 10.21.1 Detailed Description

Per-packet overhead information.

This structure is used by [ef\\_vi\\_transmit\\_alt\\_usage\(\)](#) to calculate the amount of buffering needed to store a packet. It should be filled in by [ef\\_vi\\_transmit\\_alt\\_query\\_overhead\(\)](#) or similar. Its members are not intended to be meaningful to the application and should not be obtained or interpreted in any other way.

Definition at line 720 of file ef\_vi.h.

### 10.21.2 Field Documentation

#### 10.21.2.1 mask

```
uint32_t mask
```

Rounding mask

Definition at line 724 of file ef\_vi.h.

#### 10.21.2.2 post\_round

```
uint32_t post_round
```

Bytes to add after rounding

Definition at line 726 of file ef\_vi.h.



### 10.21.2.3 pre\_round

```
uint32_t pre_round
```

Bytes to add before rounding

Definition at line 722 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.22 ef\_vi\_txq Struct Reference

TX descriptor ring.

```
#include <ef_vi.h>
```

### Data Fields

- uint32\_t [mask](#)
- void \* [descriptors](#)
- uint32\_t \* [ids](#)

### 10.22.1 Detailed Description

TX descriptor ring.

Users should not access this structure.

Definition at line 646 of file ef\_vi.h.

### 10.22.2 Field Documentation

#### 10.22.2.1 descriptors

```
void* descriptors
```

Pointer to descriptors

Definition at line 650 of file ef\_vi.h.

### 10.22.2.2 ids

```
uint32_t* ids
```

Pointer to IDs

Definition at line 652 of file ef\_vi.h.

### 10.22.2.3 mask

```
uint32_t mask
```

Mask for indexes within ring, to wrap around

Definition at line 648 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.23 ef\_vi\_txq\_state Struct Reference

State of TX descriptor ring.

```
#include <ef_vi.h>
```

### Data Fields

- [uint32\\_t previous](#)
- [uint32\\_t added](#)
- [uint32\\_t removed](#)
- [uint32\\_t ts\\_nsec](#)

### 10.23.1 Detailed Description

State of TX descriptor ring.

Users should not access this structure.

Definition at line 589 of file ef\_vi.h.

### 10.23.2 Field Documentation

#### 10.23.2.1 added

`uint32_t` added

Descriptors added to the ring

Definition at line 593 of file `ef_vi.h`.

#### 10.23.2.2 previous

`uint32_t` previous

Previous slot that has been handled

Definition at line 591 of file `ef_vi.h`.

#### 10.23.2.3 removed

`uint32_t` removed

Descriptors removed from the ring

Definition at line 595 of file `ef_vi.h`.

#### 10.23.2.4 `ts_nsec`

`uint32_t` `ts_nsec`

Timestamp in nanoseconds

Definition at line 597 of file `ef_vi.h`.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)

## 10.24 `ef_vi::ops` Struct Reference

Driver-dependent operations.

```
#include <ef_vi.h>
```

## Data Fields

- `int(* transmit)(struct ef_vi *, ef_addr base, int len, ef_request_id)`
- `int(* transmitv)(struct ef_vi *, const ef_iovec *, int iov_len, ef_request_id)`
- `int(* transmitv_init)(struct ef_vi *, const ef_iovec *, int iov_len, ef_request_id)`
- `void(* transmit_push)(struct ef_vi *)`
- `int(* transmit_pio)(struct ef_vi *, int offset, int len, ef_request_id dma_id)`
- `int(* transmit_copy_pio)(struct ef_vi *, int pio_offset, const void *src_buf, int len, ef_request_id dma_id)`
- `void(* transmit_pio_warm)(struct ef_vi *)`
- `void(* transmit_copy_pio_warm)(struct ef_vi *, int pio_offset, const void *src_buf, int len)`
- `void(* transmitv_ctpio)(struct ef_vi *, size_t frame_len, const struct iovec *iov, int iov_len, unsigned threshold)`
- `void(* transmitv_ctpio_copy)(struct ef_vi *, size_t frame_len, const struct iovec *iov, int iov_len, unsigned threshold, void *fallback)`
- `int(* transmit_alt_select)(struct ef_vi *, unsigned alt_id)`
- `int(* transmit_alt_select_default)(struct ef_vi *)`
- `int(* transmit_alt_stop)(struct ef_vi *, unsigned alt_id)`
- `int(* transmit_alt_go)(struct ef_vi *, unsigned alt_id)`
- `int(* transmit_alt_discard)(struct ef_vi *, unsigned alt_id)`
- `int(* receive_init)(struct ef_vi *, ef_addr, ef_request_id)`
- `void(* receive_push)(struct ef_vi *)`
- `int(* eventq_poll)(struct ef_vi *, ef_event *, int evs_len)`
- `void(* eventq_prime)(struct ef_vi *)`
- `void(* eventq_timer_prime)(struct ef_vi *, unsigned v)`
- `void(* eventq_timer_run)(struct ef_vi *, unsigned v)`
- `void(* eventq_timer_clear)(struct ef_vi *)`
- `void(* eventq_timer_zero)(struct ef_vi *)`

### 10.24.1 Detailed Description

Driver-dependent operations.

Definition at line 830 of file ef\_vi.h.

### 10.24.2 Field Documentation

#### 10.24.2.1 eventq\_poll

```
int(* eventq_poll)(struct ef_vi *, ef_event *, int evs_len)
```

Poll an event queue

Definition at line 879 of file ef\_vi.h.

#### 10.24.2.2 eventq\_prime

```
void(* eventq_prime) (struct ef_vi *)
```

Prime a virtual interface allowing you to go to sleep blocking on it

Definition at line 881 of file ef\_vi.h.

#### 10.24.2.3 eventq\_timer\_clear

```
void(* eventq_timer_clear) (struct ef_vi *)
```

Stop an event-queue timer

Definition at line 887 of file ef\_vi.h.

#### 10.24.2.4 eventq\_timer\_prime

```
void(* eventq_timer_prime) (struct ef_vi *, unsigned v)
```

Prime an event queue timer with a new timeout

Definition at line 883 of file ef\_vi.h.

#### 10.24.2.5 eventq\_timer\_run

```
void(* eventq_timer_run) (struct ef_vi *, unsigned v)
```

Start an event queue timer running

Definition at line 885 of file ef\_vi.h.

#### 10.24.2.6 eventq\_timer\_zero

```
void(* eventq_timer_zero) (struct ef_vi *)
```

Prime an event queue timer to expire immediately

Definition at line 889 of file ef\_vi.h.

#### 10.24.2.7 receive\_init

```
int(* receive_init) (struct ef_vi *, ef_addr, ef_request_id)
```

Initialize an RX descriptor on the RX descriptor ring

Definition at line 875 of file ef\_vi.h.

#### 10.24.2.8 receive\_push

```
void(* receive_push) (struct ef_vi *)
```

Submit newly initialized RX descriptors to the NIC

Definition at line 877 of file ef\_vi.h.

#### 10.24.2.9 transmit

```
int(* transmit) (struct ef_vi *, ef_addr base, int len, ef_request_id)
```

Transmit a packet from a single packet buffer

Definition at line 832 of file ef\_vi.h.

#### 10.24.2.10 transmit\_alt\_discard

```
int(* transmit_alt_discard) (struct ef_vi *, unsigned alt_id)
```

Transition a TX alternative to the DISCARD state

Definition at line 873 of file ef\_vi.h.

#### 10.24.2.11 transmit\_alt\_go

```
int(* transmit_alt_go) (struct ef_vi *, unsigned alt_id)
```

Transition a TX alternative to the GO state

Definition at line 871 of file ef\_vi.h.

#### 10.24.2.12 transmit\_alt\_select

```
int(* transmit_alt_select) (struct ef_vi *, unsigned alt_id)
```

Select a TX alternative as the destination for future sends

Definition at line 865 of file ef\_vi.h.

#### 10.24.2.13 transmit\_alt\_select\_default

```
int(* transmit_alt_select_default) (struct ef_vi *)
```

Select the "normal" data path as the destination for future sends

Definition at line 867 of file ef\_vi.h.

#### 10.24.2.14 transmit\_alt\_stop

```
int(* transmit_alt_stop) (struct ef_vi *, unsigned alt_id)
```

Transition a TX alternative to the STOP state

Definition at line 869 of file ef\_vi.h.

#### 10.24.2.15 transmit\_copy\_pio

```
int(* transmit_copy_pio) (struct ef_vi *, int pio_offset, const void *src_buf, int len,  
ef_request_id dma_id)
```

Copy a packet to Programmed I/O region and transmit it

Definition at line 847 of file ef\_vi.h.

#### 10.24.2.16 transmit\_copy\_pio\_warm

```
void(* transmit_copy_pio_warm) (struct ef_vi *, int pio_offset, const void *src_buf, int len)
```

Copy a packet to Programmed I/O region and warm transmit path

Definition at line 853 of file ef\_vi.h.

#### 10.24.2.17 transmit\_pio

```
int(* transmit_pio) (struct ef_vi *, int offset, int len, ef_request_id dma_id)
```

Transmit a packet already resident in Programmed I/O

Definition at line 844 of file ef\_vi.h.

#### 10.24.2.18 transmit\_pio\_warm

```
void(* transmit_pio_warm) (struct ef_vi *)
```

Warm Programmed I/O transmit path for subsequent transmit

Definition at line 851 of file ef\_vi.h.

#### 10.24.2.19 transmit\_push

```
void(* transmit_push) (struct ef_vi *)
```

Submit newly initialized TX descriptors to the NIC

Definition at line 842 of file ef\_vi.h.

#### 10.24.2.20 transmitv

```
int(* transmitv) (struct ef_vi *, const ef_iovec *, int iov_len, ef_request_id)
```

Transmit a packet from a vector of packet buffers

Definition at line 835 of file ef\_vi.h.

#### 10.24.2.21 transmitv\_ctpio

```
void(* transmitv_ctpio) (struct ef_vi *, size_t frame_len, const struct iovec *iov, int iov_len, unsigned threshold)
```

Transmit a vector of packet buffers using CTPIO

Definition at line 856 of file ef\_vi.h.



#### 10.24.2.22 transmitv\_ctpio\_copy

```
void(* transmitv_ctpio_copy) (struct ef_vi *, size_t frame_len, const struct iovec *iov, int iov_len, unsigned threshold, void *fallback)
```

Transmit a vector of packet buffers using CTPIO and copy to fallback

Definition at line 860 of file ef\_vi.h.

#### 10.24.2.23 transmitv\_init

```
int(* transmitv_init) (struct ef_vi *, const ef_iovec *, int iov_len, ef_request_id)
```

Initialize TX descriptors on the TX descriptor ring, for a vector of packet buffers

Definition at line 839 of file ef\_vi.h.

The documentation for this struct was generated from the following file:

- [ef\\_vi.h](#)



## Chapter 11

# File Documentation

### 11.1 000\_main.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

#### 11.1.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.2 005\_whats\_new.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

### 11.2.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.3 010\_overview.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

### 11.3.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.4 020\_concepts.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

### 11.4.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.5 030\_apps.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

### 11.5.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.6 040\_using.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

### 11.6.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.7 050\_examples.dox File Reference

Additional Doxygen-format documentation for [ef\\_vi](#).

### 11.7.1 Detailed Description

Additional Doxygen-format documentation for [ef\\_vi](#).

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.8 base.h File Reference

Base definitions for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/ef_vi.h>
```

## Macros

- #define `EF_VI_NIC_PAGE_SHIFT` 12  
*How much to shift an address to get the page number.*
- #define `EF_VI_NIC_PAGE_SIZE` (1<<`EF_VI_NIC_PAGE_SHIFT`)  
*The size of a page of memory on the NIC, in bytes.*
- #define `EF_ADDR_FMT` "%*CI\_PRI*x64  
*Format for outputting an ef\_addr.*
- #define `EF_INVALID_ADDR` ((`ef_addr`) -1)  
*An address that is always invalid.*

## Typedefs

- typedef int `ef_driver_handle`  
*An ef\_driver\_handle is needed to allocate resources.*

## Functions

- int `ef_eventq_wait` (`ef_vi` \*vi, `ef_driver_handle` vi\_dh, unsigned current\_ptr, const struct timeval \*timeout)  
*Block, waiting until the event queue is non-empty.*
- int `ef_driver_open` (`ef_driver_handle` \*dh\_out)  
*Obtain a driver handle.*
- int `ef_driver_close` (`ef_driver_handle` dh)  
*Close a driver handle.*

### 11.8.1 Detailed Description

Base definitions for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.8.2 Function Documentation

#### 11.8.2.1 `ef_driver_close()`

```
int ef_driver_close (  
    ef_driver_handle dh )
```

Close a driver handle.

### Parameters

<i>dh</i>	The handle to the driver to close.
-----------	------------------------------------

### Returns

0 on success, or a negative error code.

Close a driver handle.

This should be called to free up resources when the driver handle is no longer needed, but the application is to continue running.

Any associated virtual interface, protection domain, or driver structures must not be used after this call has been made.

### Note

Resources are also freed when the application exits, and so this function does not need to be called on exit.

#### 11.8.2.2 ef\_driver\_open()

```
int ef_driver_open (
    ef_driver_handle * dh_out )
```

Obtain a driver handle.

### Parameters

<i>dh_out</i>	Pointer to an ef_driver_handle, that is updated on return with the new driver handle.
---------------	---

### Returns

0 on success, or a negative error code.

Obtain a driver handle.

#### 11.8.2.3 ef\_eventq\_wait()

```
int ef_eventq_wait (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    unsigned current_ptr,
    const struct timeval * timeout )
```

Block, waiting until the event queue is non-empty.



### Parameters

<i>vi</i>	The virtual interface on which to wait.
<i>vi_dh</i>	Driver handle associated with the virtual interface.
<i>current_ptr</i>	Must come from <code>ef_eventq_current()</code> .
<i>timeout</i>	Maximum time to wait for, or 0 to wait forever.

### Returns

0 on success, or a negative error code:  
-ETIMEDOUT on time-out).

Block, waiting until the event queue is non-empty. This enables interrupts.

Note that when this function returns it is not guaranteed that an event will be present in the event queue, but in most cases there will be.

## 11.9 capabilities.h File Reference

Capabilities API for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/base.h>
```

### Enumerations

- enum `ef_vi_capability` {  
`EF_VI_CAP_PIO = 0, EF_VI_CAP_PIO_BUFFER_SIZE, EF_VI_CAP_PIO_BUFFER_COUNT,`  
`EF_VI_CAP_HW_MULTICAST_LOOPBACK,`  
`EF_VI_CAP_HW_MULTICAST_REPLICATION, EF_VI_CAP_HW_RX_TIMESTAMPING,`  
`EF_VI_CAP_HW_TX_TIMESTAMPING, EF_VI_CAP_PACKED_STREAM,`  
`EF_VI_CAP_PACKED_STREAM_BUFFER_SIZES, EF_VI_CAP_VPORTS, EF_VI_CAP_PHYS_MODE,`  
`EF_VI_CAP_BUFFER_MODE,`  
`EF_VI_CAP_MULTICAST_FILTER_CHAINING, EF_VI_CAP_MAC_SPOOFING,`  
`EF_VI_CAP_RX_FILTER_TYPE_UDP_LOCAL, EF_VI_CAP_RX_FILTER_TYPE_TCP_LOCAL,`  
`EF_VI_CAP_RX_FILTER_TYPE_UDP_FULL, EF_VI_CAP_RX_FILTER_TYPE_TCP_FULL,`  
`EF_VI_CAP_RX_FILTER_TYPE_IP_VLAN, EF_VI_CAP_RX_FILTER_TYPE_UDP6_LOCAL,`  
`EF_VI_CAP_RX_FILTER_TYPE_TCP6_LOCAL, EF_VI_CAP_RX_FILTER_TYPE_UDP6_FULL,`  
`EF_VI_CAP_RX_FILTER_TYPE_TCP6_FULL, EF_VI_CAP_RX_FILTER_TYPE_IP6_VLAN,`  
`EF_VI_CAP_RX_FILTER_TYPE_ETH_LOCAL, EF_VI_CAP_RX_FILTER_TYPE_ETH_LOCAL_VLAN,`  
`EF_VI_CAP_RX_FILTER_TYPE_UCAST_ALL, EF_VI_CAP_RX_FILTER_TYPE_MCAST_ALL,`  
`EF_VI_CAP_RX_FILTER_TYPE_UCAST_MISMATCH,`  
`EF_VI_CAP_RX_FILTER_TYPE_MCAST_MISMATCH, EF_VI_CAP_RX_FILTER_TYPE_SNIFF,`  
`EF_VI_CAP_TX_FILTER_TYPE_SNIFF,`  
`EF_VI_CAP_RX_FILTER_IP4_PROTO, EF_VI_CAP_RX_FILTER_ETHERTYPE,`  
`EF_VI_CAP_RXQ_SIZES, EF_VI_CAP_TXQ_SIZES,`  
`EF_VI_CAP_EVQ_SIZES, EF_VI_CAP_ZERO_RX_PREFIX, EF_VI_CAP_TX_PUSH_ALWAYS,`  
`EF_VI_CAP_NIC_PACE,`  
`EF_VI_CAP_RX_MERGE, EF_VI_CAP_TX_ALTERNATIVES, EF_VI_CAP_TX_ALTERNATIVES_VFIFOS,`  
`EF_VI_CAP_TX_ALTERNATIVES_CP_BUFFERS,`  
`EF_VI_CAP_RX_FW_VARIANT, EF_VI_CAP_TX_FW_VARIANT, EF_VI_CAP_CTPIO,`  
`EF_VI_CAP_TX_ALTERNATIVES_CP_BUFFER_SIZE,`  
`EF_VI_CAP_RX_FORCE_EVENT_MERGING, EF_VI_CAP_MAX }`

*Possible capabilities.*

## Functions

- int `ef_vi_capabilities_get` (`ef_driver_handle` handle, int ifindex, enum `ef_vi_capability` cap, unsigned long \*value)  
*Get the value of the given capability.*
- int `ef_vi_capabilities_max` (void)  
*Gets the maximum supported value of `ef_vi_capability`.*
- const char \* `ef_vi_capabilities_name` (enum `ef_vi_capability` cap)  
*Gets a human-readable string describing the given capability.*

### 11.9.1 Detailed Description

Capabilities API for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.9.2 Enumeration Type Documentation

#### 11.9.2.1 ef\_vi\_capability

enum `ef_vi_capability`

Possible capabilities.

#### Enumerator

<code>EF_VI_CAP_PIO</code>	Hardware capable of PIO
<code>EF_VI_CAP_PIO_BUFFER_SIZE</code>	PIO buffer size supplied to each VI
<code>EF_VI_CAP_PIO_BUFFER_COUNT</code>	Total number of PIO buffers
<code>EF_VI_CAP_HW_MULTICAST_LOOPBACK</code>	Can packets be looped back by hardware
<code>EF_VI_CAP_HW_MULTICAST_REPLICATION</code>	Can mcast be delivered to many VIs
<code>EF_VI_CAP_HW_RX_TIMESTAMPING</code>	Hardware timestamping of received packets
<code>EF_VI_CAP_HW_TX_TIMESTAMPING</code>	Hardware timestamping of transmitted packets
<code>EF_VI_CAP_PACKED_STREAM</code>	Is firmware capable of packed stream mode

Enumerator

EF_VI_CAP_PACKED_STREAM_BUFFER_SIZES	Packed stream buffer sizes supported in kB, bitmask
EF_VI_CAP_VPORTS	NIC switching, ef_pd_alloc_with_vport
EF_VI_CAP_PHYS_MODE	Is physical addressing mode supported?
EF_VI_CAP_BUFFER_MODE	Is buffer addressing mode (NIC IOMMU) supported
EF_VI_CAP_MULTICAST_FILTER_CHAINING	Chaining of multicast filters
EF_VI_CAP_MAC_SPOOFING	Can functions create filters for 'wrong' MAC addr
EF_VI_CAP_RX_FILTER_TYPE_UDP_LOCAL	Filter on local IP + UDP port
EF_VI_CAP_RX_FILTER_TYPE_TCP_LOCAL	Filter on local IP + TCP port
EF_VI_CAP_RX_FILTER_TYPE_UDP_FULL	Filter on local and remote IP + UDP port
EF_VI_CAP_RX_FILTER_TYPE_TCP_FULL	Filter on local and remote IP + TCP port
EF_VI_CAP_RX_FILTER_TYPE_IP_VLAN	Filter on any of above four types with addition of VLAN
EF_VI_CAP_RX_FILTER_TYPE_UDP6_LOCAL	Filter on local IP + UDP port
EF_VI_CAP_RX_FILTER_TYPE_TCP6_LOCAL	Filter on local IP + TCP port
EF_VI_CAP_RX_FILTER_TYPE_UDP6_FULL	Filter on local and remote IP + UDP port
EF_VI_CAP_RX_FILTER_TYPE_TCP6_FULL	Filter on local and remote IP + TCP port
EF_VI_CAP_RX_FILTER_TYPE_IP6_VLAN	Filter on any of above four types with addition of VLAN
EF_VI_CAP_RX_FILTER_TYPE_ETH_LOCAL	Filter on local MAC address
EF_VI_CAP_RX_FILTER_TYPE_ETH_LOCAL_VLAN	Filter on local MAC+VLAN
EF_VI_CAP_RX_FILTER_TYPE_UCAST_ALL	Filter on "all unicast"
EF_VI_CAP_RX_FILTER_TYPE_MCAST_ALL	Filter on "all multicast"
EF_VI_CAP_RX_FILTER_TYPE_UCAST_MISMATCH	Filter on "unicast mismatch"
EF_VI_CAP_RX_FILTER_TYPE_MCAST_MISMATCH	Filter on "multicast mismatch"
EF_VI_CAP_RX_FILTER_TYPE_SNIFF	Availability of RX sniff filters
EF_VI_CAP_TX_FILTER_TYPE_SNIFF	Availability of TX sniff filters
EF_VI_CAP_RX_FILTER_IP4_PROTO	Filter on IPv4 protocol
EF_VI_CAP_RX_FILTER_ETHERTYPE	Filter on ethertype
EF_VI_CAP_RXQ_SIZES	Available RX queue sizes, bitmask
EF_VI_CAP_TXQ_SIZES	Available TX queue sizes, bitmask
EF_VI_CAP_EVQ_SIZES	Available event queue sizes, bitmask
EF_VI_CAP_ZERO_RX_PREFIX	Availability of zero length RX packet prefix
EF_VI_CAP_TX_PUSH_ALWAYS	Is always enabling TX push supported?
EF_VI_CAP_NIC_PACE	Availability of NIC pace feature
EF_VI_CAP_RX_MERGE	Availability of RX event merging mode
EF_VI_CAP_TX_ALTERNATIVES	Availability of TX alternatives
EF_VI_CAP_TX_ALTERNATIVES_VFIFOS	Number of TX alternatives vFIFOs
EF_VI_CAP_TX_ALTERNATIVES_CP_BUFFERS	Number of TX alternatives common pool buffers
EF_VI_CAP_RX_FW_VARIANT	RX firmware variant
EF_VI_CAP_TX_FW_VARIANT	TX firmware variant
EF_VI_CAP_CTPIO	Availability of CTPIO
EF_VI_CAP_TX_ALTERNATIVES_CP_BUFFER_SIZE	Size of TX alternatives common pool buffers
EF_VI_CAP_RX_FORCE_EVENT_MERGING	RX queue is configured to force event merging
EF_VI_CAP_MAX	Maximum value of capabilities enumeration

Definition at line 63 of file capabilities.h.

## 11.9.3 Function Documentation

### 11.9.3.1 ef\_vi\_capabilities\_get()

```
int ef_vi_capabilities_get (
    ef_driver_handle handle,
    int ifindex,
    enum ef_vi_capability cap,
    unsigned long * value )
```

Get the value of the given capability.

#### Parameters

<i>handle</i>	The ef_driver_handle associated with the interface that you wish to query.
<i>ifindex</i>	The index of the interface that you wish to query. You can use if_nametoindex() to obtain this. This should be the underlying physical interface, rather than a bond, VLAN, or similar.
<i>cap</i>	The capability to get.
<i>value</i>	Pointer to location at which to store the value.

#### Returns

0 on success (capability is supported and value field is updated), or a negative error code:  
 -EOPNOTSUPP if the capability is not supported  
 -ENOSYS if the API does not know how to retrieve support for the supplied capability  
 other negative error if support could not be retrieved

### 11.9.3.2 ef\_vi\_capabilities\_max()

```
int ef_vi_capabilities_max (
    void )
```

Gets the maximum supported value of ef\_vi\_capability.

#### Returns

The maximum capability value, or a negative error code.

This function returns the maximum supported value of ef\_vi\_capability, so that all capabilities can be iterated. For example:

```
max = ef_vi_capabilities_max();
for( cap = 0; cap <= max; ++cap ) {
    rc = ef_vi_capabilities_get(driver_handle, ifindex, cap, &val);
    if( rc == 0 ) printf("%s %d\n", ef_vi_capabilities_name(cap), val);
}
```

### 11.9.3.3 ef\_vi\_capabilities\_name()

```
const char* ef_vi_capabilities_name (
    enum ef_vi_capability cap )
```

Gets a human-readable string describing the given capability.

### Parameters

<code>cap</code>	The capability for which to get the description.
------------------	--

### Returns

A string describing the capability, or a negative error code.

## 11.10 checksum.h File Reference

Checksum utility functions.

```
#include <etherfabric/base.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <ci/compat.h>
#include <ci/net/ipv6.h>
```

### Functions

- `uint32_t ef_ip_checksum` (const struct iphdr \*ip)  
*Calculate the checksum for an IP header.*
- `uint32_t ef_udp_checksum` (const struct iphdr \*ip, const struct udphdr \*udp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for a non-IPv6 UDP packet.*
- `uint32_t ef_udp_checksum_ipv6` (const ci\_ip6\_hdr \*ip6, const struct udphdr \*udp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for an IPv6 UDP packet.*
- `uint32_t ef_udp_checksum_ipx` (int af, const void \*ipx, const struct udphdr \*udp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for a UDP packet.*
- `uint32_t ef_tcp_checksum` (const struct iphdr \*ip, const struct tcphdr \*tcp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for a non-IPv6 TCP packet.*
- `uint32_t ef_tcp_checksum_ipv6` (const ci\_ip6\_hdr \*ip6, const struct tcphdr \*tcp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for an IPv6 TCP packet.*
- `uint32_t ef_tcp_checksum_ipx` (int af, const void \*ipx, const struct tcphdr \*tcp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for a TCP packet.*
- `uint32_t ef_icmpv6_checksum` (const ci\_ip6\_hdr \*ip6, const void \*icmp, const struct iovec \*iov, int iovlen)  
*Calculate the checksum for an IPv6 ICMP packet.*

## 11.10.1 Detailed Description

Checksum utility functions.

### Author

Solarflare Communications, Inc.

### Date

2018/11/06

### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.10.2 Function Documentation

### 11.10.2.1 ef\_icmpv6\_checksum()

```
uint32_t ef_icmpv6_checksum (
    const ci_ip6_hdr * ip6,
    const void * icmp,
    const struct iovec * iov,
    int iovlen )
```

Calculate the checksum for an IPv6 ICMP packet.

#### Parameters

<i>ip6</i>	The IPv6 header for the packet.
<i>icmp</i>	The ICMP header for the packet.
<i>iov</i>	Start of the iovec array describing the TCP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the ICMP packet.

Calculate the checksum for an IPv6 ICMP packet. The ICMP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

### 11.10.2.2 ef\_ip\_checksum()

```
uint32_t ef_ip_checksum (
    const struct iphdr * ip )
```

Calculate the checksum for an IP header.

#### Parameters

<i>ip</i>	The IP header to use.
-----------	-----------------------

#### Returns

The checksum of the IP header.

Calculate the checksum for an IP header. The IP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

#### 11.10.2.3 ef\_tcp\_checksum()

```
uint32_t ef_tcp_checksum (
    const struct iphdr * ip,
    const struct tcphdr * tcp,
    const struct iovec * iov,
    int iovlen )
```

Calculate the checksum for a non-IPv6 TCP packet.

#### Parameters

<i>ip</i>	The IP header for the packet.
<i>tcp</i>	The TCP header for the packet.
<i>iov</i>	Start of the iovec array describing the TCP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the TCP packet.

Calculate the checksum for a non-IPv6 TCP packet. The TCP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

#### 11.10.2.4 ef\_tcp\_checksum\_ip6()

```
uint32_t ef_tcp_checksum_ip6 (
    const ci_ip6_hdr * ip6,
    const struct tcphdr * tcp,
    const struct iovec * iov,
    int iovlen )
```

Calculate the checksum for an IPv6 TCP packet.

#### Parameters

<i>ip6</i>	The IPv6 header for the packet.
<i>tcp</i>	The TCP header for the packet.
<i>iov</i>	Start of the iovec array describing the TCP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the TCP packet.

Calculate the checksum for an IPv6 TCP packet. The TCP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

### 11.10.2.5 ef\_tcp\_checksum\_ipx()

```
uint32_t ef_tcp_checksum_ipx (
    int af,
    const void * ipx,
    const struct tcphdr * tcp,
    const struct iovec * iov,
    int iovlen )
```

Calculate the checksum for a TCP packet.

#### Parameters

<i>af</i>	The address family of the IP header for the packet.
<i>ipx</i>	The IP header for the packet.
<i>tcp</i>	The TCP header for the packet.
<i>iov</i>	Start of the iovec array describing the TCP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the TCP packet.

Calculate the checksum for a TCP packet. The TCP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

### 11.10.2.6 ef\_udp\_checksum()

```
uint32_t ef_udp_checksum (
    const struct iphdr * ip,
    const struct udphdr * udp,
    const struct iovec * iov,
    int iovlen )
```

Calculate the checksum for a non-IPv6 UDP packet.

#### Parameters

<i>ip</i>	The IP header for the packet.
<i>udp</i>	The UDP header for the packet.
<i>iov</i>	Start of the iovec array describing the UDP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the UDP packet.

Calculate the checksum for a non-IPv6 UDP packet. The UDP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

### 11.10.2.7 ef\_udp\_checksum\_ip6()

```
uint32_t ef_udp_checksum_ip6 (
    const ci_ip6_hdr * ip6,
    const struct udphdr * udp,
    const struct iovec * iov,
    int iovlen )
```

Calculate the checksum for an IPv6 UDP packet.



#### Parameters

<i>ip6</i>	The IPv6 header for the packet.
<i>udp</i>	The UDP header for the packet.
<i>iov</i>	Start of the iovec array describing the UDP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the UDP packet.

Calculate the checksum for an IPv6 UDP packet. The UDP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

#### 11.10.2.8 ef\_udp\_checksum\_ipx()

```
uint32_t ef_udp_checksum_ipx (  
    int af,  
    const void * ipx,  
    const struct udphdr * udp,  
    const struct iovec * iov,  
    int iovlen )
```

Calculate the checksum for a UDP packet.

#### Parameters

<i>af</i>	The address family of the IP header for the packet.
<i>ipx</i>	The IP header for the packet.
<i>udp</i>	The UDP header for the packet.
<i>iov</i>	Start of the iovec array describing the UDP payload.
<i>iovlen</i>	Length of the iovec array.

#### Returns

The checksum of the UDP packet.

Calculate the checksum for a UDP packet. The UDP header must be populated (with the exception of the checksum field itself, which is ignored) before calling this function.

## 11.11 ef\_vi.h File Reference

Virtual Interface definitions for EtherFabric Virtual Interface HAL.

## Data Structures

- union `ef_event`  
*A token that identifies something that has happened.*
- struct `ef_iovec`  
*ef\_iovec is similar to the standard struct iovec. An array of these is used to designate a scatter/gather list of I/O buffers.*
- struct `ef_vi_txq_state`  
*State of TX descriptor ring.*
- struct `ef_vi_rxq_state`  
*State of RX descriptor ring.*
- struct `ef_eventq_state`  
*State of event queue.*
- struct `ef_vi_txq`  
*TX descriptor ring.*
- struct `ef_vi_rxq`  
*RX descriptor ring.*
- struct `ef_vi_state`  
*State of a virtual interface.*
- struct `ef_vi_stats`  
*Statistics for a virtual interface.*
- struct `ef_vi_nic_type`  
*The type of NIC in use.*
- struct `ef_vi_transmit_alt_overhead`  
*Per-packet overhead information.*
- struct `ef_vi`  
*A virtual interface.*
- struct `ef_vi::ops`  
*Driver-dependent operations.*
- struct `ef_vi_layout_entry`  
*Layout of the data that is delivered into receive buffers.*

## Macros

- #define `EF_VI_DMA_ALIGN` 64  
*Cache line sizes for alignment purposes.*
- #define `EF_VI_MAX_QS` 32  
*The maximum number of queues per virtual interface.*
- #define `EF_VI_EVENT_POLL_MIN_EVS` 2  
*The minimum size of array to pass when polling the event queue.*
- #define `EF_REQUEST_ID_MASK` 0xffffffff  
*Mask to use with an ef\_request\_id.*
- #define `EF_EVENT_TYPE(e)` ((e).generic.type)  
*Type of event in an ef\_event e.*
- #define `EF_EVENT_RX_BYTES(e)` ((e).rx.len)  
*Get the number of bytes received.*
- #define `EF_EVENT_RX_Q_ID(e)` ((e).rx.q\_id)  
*Get the RX descriptor ring ID used for a received packet.*

- #define `EF_EVENT_RX_RQ_ID(e)` `((e).rx.rq_id)`  
*Get the dma\_id used for a received packet.*
- #define `EF_EVENT_RX_CONT(e)` `((e).rx.flags & EF_EVENT_FLAG_CONT)`  
*True if the CONTinuation Of Packet flag is set for an RX event.*
- #define `EF_EVENT_RX_SOP(e)` `((e).rx.flags & EF_EVENT_FLAG_SOP)`  
*True if the Start Of Packet flag is set for an RX event.*
- #define `EF_EVENT_RX_PS_NEXT_BUFFER(e)`  
*True if the next buffer flag is set for a packed stream event.*
- #define `EF_EVENT_RX_ISCSI_OKAY(e)` `((e).rx.flags & EF_EVENT_FLAG_ISCSI_OK)`  
*True if the iSCSIOK flag is set for an RX event.*
- #define `EF_EVENT_FLAG_SOP` 0x1  
*Start Of Packet flag.*
- #define `EF_EVENT_FLAG_CONT` 0x2  
*CONTinuation Of Packet flag.*
- #define `EF_EVENT_FLAG_ISCSI_OK` 0x4  
*iSCSI CRC validated OK flag.*
- #define `EF_EVENT_FLAG_MULTICAST` 0x8  
*Multicast flag.*
- #define `EF_EVENT_FLAG_PS_NEXT_BUFFER` 0x10  
*Packed Stream Next Buffer flag.*
- #define `EF_EVENT_FLAG_CTPIO` 0x1  
*Packets were sent successfully with CTPIO.*
- #define `EF_EVENT_TX_Q_ID(e)` `((e).tx.q_id)`  
*Get the TX descriptor ring ID used for a transmitted packet.*
- #define `EF_EVENT_TX_CTPIO(e)` `((e).tx.flags & EF_EVENT_FLAG_CTPIO)`  
*True if packets were sent successfully with CTPIO.*
- #define `EF_EVENT_RX_DISCARD_Q_ID(e)` `((e).rx_discard.q_id)`  
*Get the RX descriptor ring ID used for a discarded packet.*
- #define `EF_EVENT_RX_DISCARD_RQ_ID(e)` `((e).rx_discard.rq_id)`  
*Get the dma\_id used for a discarded packet.*
- #define `EF_EVENT_RX_DISCARD_CONT(e)` `((e).rx_discard.flags & EF_EVENT_FLAG_CONT)`  
*True if the CONTinuation Of Packet flag is set for an RX\_DISCARD event.*
- #define `EF_EVENT_RX_DISCARD_SOP(e)` `((e).rx_discard.flags & EF_EVENT_FLAG_SOP)`  
*True if the Start Of Packet flag is set for an RX\_DISCARD event.*
- #define `EF_EVENT_RX_DISCARD_TYPE(e)` `((e).rx_discard.subtype)`  
*Get the reason for an EF\_EVENT\_TYPE\_RX\_DISCARD event.*
- #define `EF_EVENT_RX_DISCARD_BYTES(e)` `((e).rx_discard.len)`  
*Get the length of a discarded packet.*
- #define `EF_EVENT_RX_MULTI_Q_ID(e)` `((e).rx_multi.q_id)`  
*Get the RX descriptor ring ID used for a received packet.*
- #define `EF_EVENT_RX_MULTI_CONT(e)`  
*True if the CONTinuation Of Packet flag is set for an RX HT event.*
- #define `EF_EVENT_RX_MULTI_SOP(e)`  
*True if the Start Of Packet flag is set for an RX HT event.*
- #define `EF_EVENT_RX_MULTI_DISCARD_TYPE(e)` `((e).rx_multi_discard.subtype)`  
*Get the reason for an EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD event.*
- #define `EF_EVENT_TX_ERROR_Q_ID(e)` `((e).tx_error.q_id)`  
*Get the TX descriptor ring ID used for a transmit error.*
- #define `EF_EVENT_TX_ERROR_TYPE(e)` `((e).tx_error.subtype)`

- Get the reason for a TX\_ERROR event.*

  - #define EF\_VI\_SYNC\_FLAG\_CLOCK\_SET 1
    - The adapter clock has previously been set in sync with the system.*
  - #define EF\_VI\_SYNC\_FLAG\_CLOCK\_IN\_SYNC 2
    - The adapter clock is in sync with the external clock (PTP)*
  - #define EF\_EVENT\_TX\_WITH\_TIMESTAMP\_Q\_ID(e) ((e).tx\_timestamp.q\_id)
    - Get the TX descriptor ring ID used for a timestamped packet.*
  - #define EF\_EVENT\_TX\_WITH\_TIMESTAMP\_RQ\_ID(e) ((e).tx\_timestamp.rq\_id)
    - Get the dma\_id used for a timestamped packet.*
  - #define EF\_EVENT\_TX\_WITH\_TIMESTAMP\_SEC(e) ((e).tx\_timestamp.ts\_sec)
    - Get the number of seconds from the timestamp of a transmitted packet.*
  - #define EF\_EVENT\_TX\_WITH\_TIMESTAMP\_NSEC(e) ((e).tx\_timestamp.ts\_nsec)
    - Get the number of nanoseconds from the timestamp of a transmitted packet.*
  - #define EF\_EVENT\_TX\_WITH\_TIMESTAMP\_SYNC\_MASK (EF\_VI\_SYNC\_FLAG\_CLOCK\_SET | EF\_VI\_SYNC\_FLAG\_CLOCK\_IN\_SYNC)
    - Mask for the sync flags in the timestamp of a transmitted packet.*
  - #define EF\_EVENT\_TX\_WITH\_TIMESTAMP\_SYNC\_FLAGS(e) ((e).tx\_timestamp.ts\_nsec & EF\_EVENT\_TX\_WITH\_TIMESTAMP\_SYNC\_MASK)
    - Get the sync flags from the timestamp of a transmitted packet.*
  - #define EF\_EVENT\_TX\_ALT\_Q\_ID(e) ((e).tx\_alt.q\_id)
    - Get the TX descriptor ring ID used for a TX alternative packet.*
  - #define EF\_EVENT\_TX\_ALT\_ALT\_ID(e) ((e).tx\_alt.alt\_id)
    - Get the TX alternative ID used for a TX alternative packet.*
  - #define EF\_EVENT\_RX\_NO\_DESC\_TRUNC\_Q\_ID(e) ((e).rx\_no\_desc\_trunc.q\_id)
    - Get the RX descriptor ring ID used for a received packet that was truncated due to a lack of descriptors.*
  - #define EF\_EVENT\_SW\_DATA\_MASK 0xffff
    - Mask for the data in a software generated event.*
  - #define EF\_EVENT\_SW\_DATA(e) ((e).sw.data)
    - Get the data for an EF\_EVENT\_TYPE\_SW event.*
  - #define EF\_EVENT\_FMT "[ev:%x]"
    - Output format for an ef\_event.*
  - #define EF\_EVENT\_PRI\_ARG(e) (unsigned) (e).generic.type
    - Get the type of an event.*
  - #define ef\_vi\_receive\_init(vi, addr, dma\_id) (vi)->ops.receive\_init((vi), (addr), (dma\_id))
    - Initialize an RX descriptor on the RX descriptor ring.*
  - #define ef\_vi\_receive\_push(vi) (vi)->ops.receive\_push((vi))
    - Submit newly initialized RX descriptors to the NIC.*
  - #define EF\_VI\_RECEIVE\_BATCH 15
    - Maximum number of receive completions per receive event.*
  - #define ef\_vi\_transmitv\_init(vi, iov, iov\_len, dma\_id) (vi)->ops.transmitv\_init((vi), (iov), (iov\_len), (dma\_id))
    - Initialize TX descriptors on the TX descriptor ring, for a vector of packet buffers.*
  - #define ef\_vi\_transmit\_push(vi) (vi)->ops.transmit\_push((vi))
    - Submit newly initialized TX descriptors to the NIC.*
  - #define ef\_vi\_transmit(vi, base, len, dma\_id) (vi)->ops.transmit((vi), (base), (len), (dma\_id))
    - Transmit a packet from a single packet buffer.*
  - #define ef\_vi\_transmitv(vi, iov, iov\_len, dma\_id) (vi)->ops.transmitv((vi), (iov), (iov\_len), (dma\_id))
    - Transmit a packet from a vector of packet buffers.*
  - #define ef\_vi\_transmit\_pio(vi, offset, len, dma\_id) (vi)->ops.transmit\_pio((vi), (offset), (len), (dma\_id))
    - Transmit a packet already resident in Programmed I/O.*

- #define `ef_vi_transmit_copy_pio`(vi, pio\_offset, src\_buf, len, dma\_id)  
*Transmit a packet by copying it into the Programmed I/O region.*
- #define `ef_vi_transmit_pio_warm`(vi) (vi)->ops.transmit\_pio\_warm((vi))  
*Warm Programmed I/O transmit path for subsequent transmit.*
- #define `ef_vi_transmit_copy_pio_warm`(vi, pio\_offset, src\_buf, len) (vi)->ops.transmit\_copy\_pio\_warm((vi), (pio\_offset), (src\_buf), (len))  
*Copy a packet to Programmed I/O region and warm transmit path.*
- #define `EF_VI_TRANSMIT_BATCH` 64  
*Maximum number of transmit completions per transmit event.*
- #define `ef_vi_transmit_alt_select`(vi, alt\_id) (vi)->ops.transmit\_alt\_select((vi), (alt\_id))  
*Select a TX alternative as the destination for future sends.*
- #define `ef_vi_transmit_alt_select_normal`(vi) (vi)->ops.transmit\_alt\_select\_default((vi))  
*Select the "normal" data path as the destination for future sends.*
- #define `ef_vi_transmit_alt_stop`(vi, alt\_id) (vi)->ops.transmit\_alt\_stop((vi), (alt\_id))  
*Transition a TX alternative to the STOP state.*
- #define `ef_vi_transmit_alt_go`(vi, alt\_id) (vi)->ops.transmit\_alt\_go((vi), (alt\_id))  
*Transition a TX alternative to the GO state.*
- #define `ef_vi_transmit_alt_discard`(vi, alt\_id) (vi)->ops.transmit\_alt\_discard((vi), (alt\_id))  
*Transition a TX alternative to the DISCARD state.*
- #define `ef_vi_transmitv_ctpio`(vi, frame\_len, frame\_iov, frame\_iov\_len, ct\_threshold)  
*Transmit a packet using CTPIO from an array of buffers.*
- #define `ef_vi_transmitv_ctpio_copy`(vi, frame\_len, frame\_iov, frame\_iov\_len, ct\_threshold, fallback)  
*Transmit a packet using CTPIO from an array of buffers, simultaneously copying the data into a fallback buffer.*
- #define `EF_VI_CTPIO_CT_THRESHOLD_SNF` 0xffff
- #define `ef_eventq_prime`(vi) (vi)->ops.eventq\_prime((vi))  
*Prime a virtual interface allowing you to go to sleep blocking on it.*
- #define `ef_eventq_poll`(evq, evs, evs\_len) (evq)->ops.eventq\_poll((evq), (evs), (evs\_len))  
*Poll an event queue.*

## Typedefs

- typedef uint32\_t `ef_eventq_ptr`  
*A pointer to an event queue.*
- typedef uint64\_t `ef_addr`  
*An address.*
- typedef char \* `ef_vi_ioaddr_t`  
*An address of an I/O area for a virtual interface.*
- typedef int `ef_request_id`  
*A DMA request identifier.*
- typedef struct `ef_vi` `ef_vi`  
*A virtual interface.*

## Enumerations

- enum {  
EF\_EVENT\_TYPE\_RX, EF\_EVENT\_TYPE\_TX, EF\_EVENT\_TYPE\_RX\_DISCARD,  
EF\_EVENT\_TYPE\_TX\_ERROR,  
EF\_EVENT\_TYPE\_RX\_NO\_DESC\_TRUNC, EF\_EVENT\_TYPE\_SW, EF\_EVENT\_TYPE\_OFLOW,  
EF\_EVENT\_TYPE\_TX\_WITH\_TIMESTAMP,  
EF\_EVENT\_TYPE\_RX\_PACKED\_STREAM, EF\_EVENT\_TYPE\_RX\_MULTI, EF\_EVENT\_TYPE\_TX\_ALT,  
EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD }  
*Possible types of events.*
- enum {  
EF\_EVENT\_RX\_DISCARD\_CSUM\_BAD, EF\_EVENT\_RX\_DISCARD\_MCAST\_MISMATCH,  
EF\_EVENT\_RX\_DISCARD\_CRC\_BAD, EF\_EVENT\_RX\_DISCARD\_TRUNC,  
EF\_EVENT\_RX\_DISCARD\_RIGHTS, EF\_EVENT\_RX\_DISCARD\_EV\_ERROR,  
EF\_EVENT\_RX\_DISCARD\_OTHER, EF\_EVENT\_RX\_DISCARD\_INNER\_CSUM\_BAD,  
EF\_EVENT\_RX\_DISCARD\_MAX }  
*The reason for an EF\_EVENT\_TYPE\_RX\_DISCARD event.*
- enum { EF\_EVENT\_TX\_ERROR\_RIGHTS, EF\_EVENT\_TX\_ERROR\_OFLOW,  
EF\_EVENT\_TX\_ERROR\_2BIG, EF\_EVENT\_TX\_ERROR\_BUS }  
*The reason for an EF\_EVENT\_TYPE\_TX\_ERROR event.*
- enum ef\_vi\_flags {  
EF\_VI\_FLAGS\_DEFAULT = 0x0, EF\_VI\_ISCSI\_RX\_HDIG = 0x2, EF\_VI\_ISCSI\_TX\_HDIG = 0x4,  
EF\_VI\_ISCSI\_RX\_DDIG = 0x8,  
EF\_VI\_ISCSI\_TX\_DDIG = 0x10, EF\_VI\_TX\_PHYS\_ADDR = 0x20, EF\_VI\_RX\_PHYS\_ADDR = 0x40,  
EF\_VI\_TX\_IP\_CSUM\_DIS = 0x80,  
EF\_VI\_TX\_TCPUDP\_CSUM\_DIS = 0x100, EF\_VI\_TX\_TCPUDP\_ONLY = 0x200, EF\_VI\_TX\_FILTER\_IP =  
0x400, EF\_VI\_TX\_FILTER\_MAC = 0x800,  
EF\_VI\_TX\_FILTER\_MASK\_1 = 0x1000, EF\_VI\_TX\_FILTER\_MASK\_2 = 0x2000,  
EF\_VI\_TX\_FILTER\_MASK\_3 = (0x1000 | 0x2000), EF\_VI\_TX\_PUSH\_DISABLE = 0x4000,  
EF\_VI\_TX\_PUSH\_ALWAYS = 0x8000, EF\_VI\_RX\_TIMESTAMPS = 0x10000, EF\_VI\_TX\_TIMESTAMPS =  
0x20000, EF\_VI\_RX\_PACKED\_STREAM = 0x80000,  
EF\_VI\_RX\_PS\_BUF\_SIZE\_64K = 0x100000, EF\_VI\_RX\_EVENT\_MERGE = 0x200000, EF\_VI\_TX\_ALT =  
0x400000, EF\_VI\_ENABLE\_EV\_TIMER = 0x800000,  
EF\_VI\_TX\_CTPIO = 0x1000000, EF\_VI\_TX\_CTPIO\_NO\_POISON = 0x2000000 }  
*Flags that can be requested when allocating an ef\_vi.*
- enum ef\_vi\_out\_flags { EF\_VI\_OUT\_CLOCK\_SYNC\_STATUS = 0x1 }  
*Flags that can be returned when an ef\_vi has been allocated.*
- enum ef\_vi\_rx\_discard\_err\_flags {  
EF\_VI\_DISCARD\_RX\_L4\_CSUM\_ERR = 0x1, EF\_VI\_DISCARD\_RX\_L3\_CSUM\_ERR = 0x2,  
EF\_VI\_DISCARD\_RX\_ETH\_FCS\_ERR = 0x4, EF\_VI\_DISCARD\_RX\_ETH\_LEN\_ERR = 0x8,  
EF\_VI\_DISCARD\_RX\_TOBE\_DISC = 0x10, EF\_VI\_DISCARD\_RX\_INNER\_L4\_CSUM\_ERR = 0x20,  
EF\_VI\_DISCARD\_RX\_INNER\_L3\_CSUM\_ERR = 0x40 }  
*Flags that define which errors will cause RX\_DISCARD events.*
- enum ef\_timestamp\_format { TS\_FORMAT\_SECONDS\_27FRACTION = 0,  
TS\_FORMAT\_SECONDS\_QTR\_NANOSECONDS = 1 }  
*Timestamp formats supported by various cards.*
- enum ef\_vi\_arch { EF\_VI\_ARCH\_FALCON, EF\_VI\_ARCH\_EF10 }  
*NIC architectures that are supported.*
- enum ef\_vi\_layout\_type { EF\_VI\_LAYOUT\_FRAME, EF\_VI\_LAYOUT\_MINOR\_TICKS,  
EF\_VI\_LAYOUT\_PACKET\_LENGTH }  
*Types of layout that are used for receive buffers.*

## Functions

- `ef_vi_inline unsigned ef_vi_resource_id (ef_vi *vi)`  
*Return the resource ID of the virtual interface.*
- `ef_vi_inline enum ef_vi_flags ef_vi_flags (ef_vi *vi)`  
*Return the flags of the virtual interface.*
- `ef_vi_inline unsigned ef_vi_instance (ef_vi *vi)`  
*Return the instance ID of the virtual interface.*
- `const char * ef_vi_version_str (void)`  
*Return a string that identifies the version of ef\_vi.*
- `const char * ef_vi_driver_interface_str (void)`  
*Returns a string that identifies the char driver interface required.*
- `ef_vi_inline int ef_vi_receive_prefix_len (const ef_vi *vi)`  
*Returns the length of the prefix at the start of a received packet.*
- `ef_vi_inline int ef_vi_receive_buffer_len (const ef_vi *vi)`  
*Returns the length of a receive buffer.*
- `ef_vi_inline void ef_vi_receive_set_buffer_len (ef_vi *vi, unsigned buf_len)`  
*Sets the length of receive buffers.*
- `ef_vi_inline int ef_vi_receive_space (const ef_vi *vi)`  
*Returns the amount of free space in the RX descriptor ring.*
- `ef_vi_inline int ef_vi_receive_fill_level (const ef_vi *vi)`  
*Returns the fill level of the RX descriptor ring.*
- `ef_vi_inline int ef_vi_receive_capacity (const ef_vi *vi)`  
*Returns the total capacity of the RX descriptor ring.*
- `int ef_vi_receive_post (ef_vi *vi, ef_addr addr, ef_request_id dma_id)`  
*Initialize an RX descriptor on the RX descriptor ring, and submit it to the NIC.*
- `int ef_vi_receive_get_timestamp (ef_vi *vi, const void *pkt, struct timespec *ts_out)`  
*Deprecated: use ef\_vi\_receive\_get\_timestamp\_with\_sync\_flags() instead.*
- `int ef_vi_receive_get_timestamp_with_sync_flags (ef_vi *vi, const void *pkt, struct timespec *ts_out, unsigned *flags_out)`  
*Retrieve the UTC timestamp associated with a received packet, and the clock sync status flags.*
- `int ef_vi_receive_get_bytes (ef_vi *vi, const void *pkt, uint16_t *bytes_out)`  
*Retrieve the number of bytes in a received packet in RX event merge mode.*
- `int ef_vi_receive_unbundle (ef_vi *ep, const ef_event *event, ef_request_id *ids)`  
*Unbundle an event of type EF\_EVENT\_TYPE\_RX\_MULTI or EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD.*
- `int ef_vi_receive_set_discards (ef_vi *vi, unsigned discard_err_flags)`  
*Set which errors cause an EF\_EVENT\_TYPE\_RX\_DISCARD event.*
- `ef_vi_inline int ef_vi_transmit_space (const ef_vi *vi)`  
*Returns the amount of free space in the TX descriptor ring.*
- `ef_vi_inline int ef_vi_transmit_fill_level (const ef_vi *vi)`  
*Returns the fill level of the TX descriptor ring.*
- `ef_vi_inline int ef_vi_transmit_capacity (const ef_vi *vi)`  
*Returns the total capacity of the TX descriptor ring.*
- `int ef_vi_transmit_init (ef_vi *vi, ef_addr addr, int bytes, ef_request_id dma_id)`  
*Initialize a TX descriptor on the TX descriptor ring, for a single packet buffer.*
- `void ef_vi_transmit_init_undo (ef_vi *vi)`  
*Remove all TX descriptors from the TX descriptor ring that have been initialized since last transmit.*
- `int ef_vi_transmit_unbundle (ef_vi *ep, const ef_event *event, ef_request_id *ids)`  
*Unbundle an event of type of type EF\_EVENT\_TYPE\_TX or EF\_EVENT\_TYPE\_TX\_ERROR.*

- unsigned `ef_vi_transmit_alt_num_ids` (`ef_vi *vi`)  
*Return the number of TX alternatives allocated for a virtual interface.*
- int `ef_vi_transmit_alt_query_overhead` (`ef_vi *vi`, struct `ef_vi_transmit_alt_overhead *params`)  
*Query per-packet overhead parameters.*
- `ef_vi_inline` `ef_vi_pure` `uint32_t ef_vi_transmit_alt_usage` (`const struct ef_vi_transmit_alt_overhead *params`, `uint32_t pkt_len`)  
*Calculate a packet's buffer usage.*
- void `ef_vi_set_tx_push_threshold` (`ef_vi *vi`, unsigned threshold)  
*Set the threshold at which to switch from using TX descriptor push to using a doorbell.*
- `ef_vi_inline` void `ef_vi_transmit_ctpio` (`ef_vi *vi`, `const void *frame_buf`, `size_t frame_len`, unsigned `ct_threshold`)  
*Transmit a packet using CTPIO.*
- int `ef_vi_transmit_ctpio_fallback` (`ef_vi *vi`, `ef_addr dma_addr`, `size_t len`, `ef_request_id dma_id`)  
*Post fallback frame for a CTPIO transmit.*
- int `ef_vi_transmitv_ctpio_fallback` (`ef_vi *vi`, `const ef_iovec *dma_iov`, `int dma_iov_len`, `ef_request_id dma_id`)  
*Post fallback frame for a CTPIO transmit.*
- int `ef_eventq_has_event` (`const ef_vi *vi`)  
*Returns true if `ef_eventq_poll()` will return event(s)*
- int `ef_eventq_has_many_events` (`const ef_vi *evq`, `int n_events`)  
*Returns true if there are a given number of events in the event queue.*
- int `ef_eventq_capacity` (`ef_vi *vi`)  
*Returns the capacity of an event queue.*
- `ef_vi_inline` unsigned `ef_eventq_current` (`ef_vi *evq`)  
*Get the current offset into the event queue.*
- int `ef_vi_receive_query_layout` (`ef_vi *vi`, `const ef_vi_layout_entry **const layout_out`, `int *layout_len_out`)  
*Gets the layout of the data that the adapter delivers into receive buffers.*

### 11.11.1 Detailed Description

Virtual Interface definitions for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.11.2 Macro Definition Documentation



### 11.11.2.1 EF\_EVENT\_RX\_MULTI\_CONT

```
#define EF_EVENT_RX_MULTI_CONT(  
    e )
```

**Value:**

```
((e).rx_multi.flags & \
```

[EF\\_EVENT\\_FLAG\\_CONT](#))

True if the CONTinuation Of Packet flag is set for an RX HT event.

Definition at line 351 of file ef\_vi.h.

### 11.11.2.2 EF\_EVENT\_RX\_MULTI\_SOP

```
#define EF_EVENT_RX_MULTI_SOP(  
    e )
```

**Value:**

```
((e).rx_multi.flags & \
```

[EF\\_EVENT\\_FLAG\\_SOP](#))

True if the Start Of Packet flag is set for an RX HT event.

Definition at line 354 of file ef\_vi.h.

### 11.11.2.3 EF\_EVENT\_RX\_PS\_NEXT\_BUFFER

```
#define EF_EVENT_RX_PS_NEXT_BUFFER(  
    e )
```

**Value:**

```
((e).rx_packed_stream.flags & \
```

[EF\\_EVENT\\_FLAG\\_PS\\_NEXT\\_BUFFER](#))

True if the next buffer flag is set for a packed stream event.

Definition at line 305 of file ef\_vi.h.

### 11.11.2.4 ef\_eventq\_poll

```
#define ef_eventq_poll(  
    evq,  
    evs,  
    evs_len ) (evq)->ops.eventq_poll((evq), (evs), (evs_len))
```

Poll an event queue.

### Parameters

<i>evq</i>	The event queue to poll.
<i>evs</i>	Array in which to return polled events.
<i>evs_len</i>	Length of the <i>evs</i> array, must be $\geq$ EF_VI_EVENT_POLL_MIN_EVS.

### Returns

The number of events retrieved.

Poll an event queue. Any events that have been raised are added to the given array. Most events correspond to packets arriving, or packet transmission completing. This function is critical to latency, and must be called as often as possible.

This function returns immediately, even if there are no outstanding events. The array might not be full on return.

Definition at line 2092 of file ef\_vi.h.

#### 11.11.2.5 ef\_eventq\_prime

```
#define ef_eventq_prime(  
    vi ) (vi)->ops.eventq_prime((vi))
```

Prime a virtual interface allowing you to go to sleep blocking on it.

### Parameters

<i>vi</i>	The virtual interface to prime.
-----------	---------------------------------

### Returns

None.

Prime a virtual interface allowing you to go to sleep blocking on it.

Definition at line 2072 of file ef\_vi.h.

#### 11.11.2.6 EF\_VI\_CTPIO\_CT\_THRESHOLD\_SNF

```
#define EF_VI_CTPIO_CT_THRESHOLD_SNF 0xffff
```

Cut-through threshold to use if store-and-forward behavior is wanted for all packet sizes. For use with [ef\\_vi\\_transmit\\_ctpio\(\)](#) and [ef\\_vi\\_transmitv\\_ctpio\(\)](#).

Definition at line 2025 of file ef\_vi.h.

#### 11.11.2.7 ef\_vi\_receive\_init

```
#define ef_vi_receive_init(  
    vi,  
    addr,  
    dma_id ) (vi)->ops.receive_init((vi), (addr), (dma_id))
```

Initialize an RX descriptor on the RX descriptor ring.

#### Parameters

<i>vi</i>	The virtual interface for which to initialize an RX descriptor.
<i>addr</i>	DMA address of the packet buffer to associate with the descriptor, as obtained from <a href="#">ef_memreg_dma_addr()</a> .
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

#### Returns

0 on success, or a negative error code.

Initialize an RX descriptor on the RX descriptor ring, and prepare the associated packet buffer (identified by its DMA address) to receive packets. This function only writes a few bytes into host memory, and is very fast.

Definition at line 1103 of file ef\_vi.h.

#### 11.11.2.8 ef\_vi\_receive\_push

```
#define ef_vi_receive_push(  
    vi ) (vi)->ops.receive_push((vi))
```

Submit newly initialized RX descriptors to the NIC.

#### Parameters

<i>vi</i>	The virtual interface for which to push descriptors.
-----------	--

#### Returns

None.

Submit newly initialized RX descriptors to the NIC. The NIC can then receive packets into the associated packet buffers.

For Solarflare 7000-series NICs, this function submits RX descriptors only in multiples of 8. This is to conform with hardware requirements. If the number of newly initialized RX descriptors is not exactly divisible by 8, this function does not submit any remaining descriptors (up to 7 of them).

Definition at line 1122 of file ef\_vi.h.

#### 11.11.2.9 ef\_vi\_transmit

```
#define ef_vi_transmit(  
    vi,  
    base,  
    len,  
    dma_id ) (vi)->ops.transmit((vi), (base), (len), (dma_id))
```

Transmit a packet from a single packet buffer.

### Parameters

<i>vi</i>	The virtual interface for which to initialize and push a TX descriptor.
<i>base</i>	DMA address of the packet buffer to associate with the descriptor, as obtained from <a href="#">ef_memreg_dma_addr()</a> .
<i>len</i>	The size of the packet to transmit.
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

### Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

Transmit a packet from a single packet buffer. This initializes a TX descriptor on the TX descriptor ring, and submits it to the NIC. The NIC can then transmit a packet from the associated packet buffer.

This function simply wraps [ef\\_vi\\_transmit\\_init\(\)](#) and [ef\\_vi\\_transmit\\_push\(\)](#). It is provided as a convenience. It is less efficient than submitting the descriptors in batches by calling the functions separately, but unless there is a batch of packets to transmit, calling this function is often the right thing to do.

Definition at line 1464 of file [ef\\_vi.h](#).

#### 11.11.2.10 ef\_vi\_transmit\_alt\_discard

```
#define ef_vi_transmit_alt_discard(  
    vi,  
    alt_id ) (vi)->ops.transmit_alt_discard((vi), (alt_id))
```

Transition a TX alternative to the DISCARD state.

### Parameters

<i>vi</i>	The virtual interface associated with the TX alternative.
<i>alt_id</i>	The TX alternative to transition to the DISCARD state.

### Returns

0 on success, or a negative error code.

Transitions a TX alternative to the DISCARD state. Packets buffered in the alternative are discarded.

As packets are discarded, events of type `EF_EVENT_TYPE_TX_ALT` are returned to the application. The application should normally wait until all packets have been discarded before transitioning to a different state.

Memory for the TX alternative remains allocated, and is not freed until the virtual interface is freed.

Definition at line 1788 of file [ef\\_vi.h](#).

#### 11.11.2.11 ef\_vi\_transmit\_alt\_go

```
#define ef_vi_transmit_alt_go(  
    vi,  
    alt_id ) (vi)->ops.transmit_alt_go((vi), (alt_id))
```

Transition a TX alternative to the GO state.

#### Parameters

<i>vi</i>	The virtual interface associated with the TX alternative.
<i>alt_id</i>	The TX alternative to transition to the GO state.

#### Returns

0 on success, or a negative error code.

Transitions a TX alternative to the GO state. Packets buffered in the alternative are transmitted to the network.

As packets are transmitted events of type `EF_EVENT_TYPE_TX_ALT` are returned to the application. The application should normally wait until all packets have been sent before transitioning to a different state.

Definition at line 1767 of file `ef_vi.h`.

#### 11.11.2.12 ef\_vi\_transmit\_alt\_select

```
#define ef_vi_transmit_alt_select(  
    vi,  
    alt_id ) (vi)->ops.transmit_alt_select((vi), (alt_id))
```

Select a TX alternative as the destination for future sends.

#### Parameters

<i>vi</i>	The virtual interface associated with the TX alternative.
<i>alt_id</i>	The TX alternative to select.

#### Returns

0 on success, or a negative error code.

Selects a TX alternative as the destination for future sends. Packets can be sent to it using normal send calls such as [ef\\_vi\\_transmit\(\)](#). The action then taken depends on the state of the TX alternative:

- if the TX alternative is in the STOP state, the packet is buffered for possible future transmission
- if the TX alternative is in the GO state, the packet is immediately transmitted.

Definition at line 1723 of file `ef_vi.h`.

#### 11.11.2.13 ef\_vi\_transmit\_alt\_select\_normal

```
#define ef_vi_transmit_alt_select_normal(  
    vi ) (vi)->ops.transmit_alt_select_default((vi))
```

Select the "normal" data path as the destination for future sends.

## Parameters

<i>vi</i>	A virtual interface associated with a TX alternative.
-----------	---

Selects the "normal" data path as the destination for future sends. The virtual interface then transmits packets to the network immediately, in the normal way. This call undoes the effect of [ef\\_vi\\_transmit\\_alt\\_select\(\)](#).

Definition at line 1737 of file ef\_vi.h.

### 11.11.2.14 ef\_vi\_transmit\_alt\_stop

```
#define ef_vi_transmit_alt_stop(  
    vi,  
    alt_id ) (vi)->ops.transmit_alt_stop((vi), (alt_id))
```

Transition a TX alternative to the STOP state.

## Parameters

<i>vi</i>	The virtual interface associated with the TX alternative.
<i>alt_id</i>	The TX alternative to transition to the STOP state.

Transitions a TX alternative to the STOP state. Packets that are sent to a TX alternative in the STOP state are buffered on the adapter.

Definition at line 1749 of file ef\_vi.h.

### 11.11.2.15 ef\_vi\_transmit\_copy\_pio

```
#define ef_vi_transmit_copy_pio(  
    vi,  
    pio_offset,  
    src_buf,  
    len,  
    dma_id )
```

## Value:

```
(vi)->ops.transmit_copy_pio((vi), (pio_offset), (src_buf),  
                            (len), (dma_id)) \
```

Transmit a packet by copying it into the Programmed I/O region.

## Parameters

<i>vi</i>	The virtual interface from which to transmit.
<i>pio_offset</i>	The offset within its Programmed I/O region to the start of the packet. This must be aligned to at least a 64-byte boundary.
<i>src_buf</i>	The source buffer from which to read the packet.
<i>len</i>	Length of the packet to transmit. This must be at least 16 bytes.
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

## Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

Transmit a packet by copying it into the Programmed I/O region.

The `src_buf` parameter must point at a complete packet that is copied to the adapter and transmitted. The source buffer need not be registered, and is available for re-use immediately after this call returns.

This call does not copy the packet data into the local copy of the adapter's Programmed I/O buffer. As a result it is slightly faster than calling `ef_pio_memcpy()` followed by `ef_vi_transmit_pio()`.

The Programmed I/O region used by this call must not be reused until an event indicating TX completion is handled (see [Transmitting Packets](#)), thus completing the transmit operation for the packet. Failure to do so might corrupt an ongoing transmit.

The Programmed I/O region can hold multiple smaller packets, referenced by different offset parameters. All other constraints must still be observed, including:

- alignment
- minimum size
- maximum size
- avoiding reuse until transmission is complete.

Definition at line 1579 of file `ef_vi.h`.

### 11.11.2.16 ef\_vi\_transmit\_copy\_pio\_warm

```
#define ef_vi_transmit_copy_pio_warm(  
    vi,  
    pio_offset,  
    src_buf,  
    len ) (vi)->ops.transmit_copy_pio_warm((vi), (pio_offset), (src_buf), (len))
```

Copy a packet to Programmed I/O region and warm transmit path.

#### Parameters

<i>vi</i>	The virtual interface from which transmit is planned.
<i>pio_offset</i>	The offset within its Programmed I/O region to the start of the packet. This must be aligned to at least a 64-byte boundary.
<i>src_buf</i>	The source buffer from which to read the packet.
<i>len</i>	Length of the packet to transmit. This must be at least 16 bytes.

## Returns

None

Copy a packet to Programmed I/O region and warm transmit path

The application can call this function in advance of calls to [ef\\_vi\\_transmit\\_copy\\_pio\(\)](#) to reduce latency jitter caused by code and state being evicted from cache during delays between transmits. This is also effective before the first transmit using Programmed I/O.

No data is sent but this function will copy data to the Programmed I/O region. Therefore all constraints regarding copying to the Programmed I/O region must be met. This includes not reusing a region previously transmitted from until the corresponding TX completion has been handled. See [ef\\_vi\\_transmit\\_copy\\_pio\(\)](#) for full details of the constraints.

While this may also benefit a subsequent call to [ef\\_vi\\_transmit\\_pio\(\)](#), it follows a different code path. See [ef\\_vi\\_transmit\\_pio\\_warm\(\)](#) for a warming function designed to warm for [ef\\_vi\\_transmit\\_pio\(\)](#).

Definition at line 1635 of file ef\_vi.h.

### 11.11.2.17 ef\_vi\_transmit\_pio

```
#define ef_vi_transmit_pio(  
    vi,  
    offset,  
    len,  
    dma_id ) (vi)->ops.transmit_pio((vi), (offset), (len), (dma_id))
```

Transmit a packet already resident in Programmed I/O.

#### Parameters

<i>vi</i>	The virtual interface from which to transmit.
<i>offset</i>	The offset within its Programmed I/O region to the start of the packet. This must be aligned to at least a 64-byte boundary.
<i>len</i>	Length of the packet to transmit. This must be at least 16 bytes.
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

## Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

Transmit a packet already resident in Programmed I/O.

The Programmed I/O region used by this call must not be reused until an event indicating TX completion is handled (see [Transmitting Packets](#)), thus completing the transmit operation for the packet. Failure to do so might corrupt an ongoing transmit.

The Programmed I/O region can hold multiple packets, referenced by different offset parameters. All other constraints must still be observed, including:



- alignment
- minimum size
- maximum size
- avoiding reuse until transmission is complete.

Definition at line 1536 of file ef\_vi.h.

#### 11.11.2.18 ef\_vi\_transmit\_pio\_warm

```
#define ef_vi_transmit_pio_warm(  
    vi ) (vi)->ops.transmit_pio_warm((vi))
```

Warm Programmed I/O transmit path for subsequent transmit.

##### Parameters

<i>vi</i>	The virtual interface from which transmit is planned.
-----------	---

##### Returns

None

Warm Programmed I/O transmit path for a subsequent transmit.

The application can call this function in advance of calls to [ef\\_vi\\_transmit\\_pio\(\)](#) to reduce latency jitter caused by code and state being evicted from cache during delays between transmits. This is also effective before the first transmit using Programmed I/O.

While this may also benefit a subsequent call to [ef\\_vi\\_transmit\\_copy\\_pio\(\)](#), it follows a different code path. See [ef\\_vi\\_transmit\\_copy\\_pio\\_warm\(\)](#) for a warming function designed to warm for [ef\\_vi\\_transmit\\_copy\\_pio\(\)](#).

Definition at line 1601 of file ef\_vi.h.

#### 11.11.2.19 ef\_vi\_transmit\_push

```
#define ef_vi_transmit_push(  
    vi ) (vi)->ops.transmit_push((vi))
```

Submit newly initialized TX descriptors to the NIC.

##### Parameters

<i>vi</i>	The virtual interface for which to push descriptors.
-----------	--

##### Returns

None.

Submit newly initialized TX descriptors to the NIC. The NIC can then transmit packets from the associated packet buffers.

New TX descriptors must have been initialized using [ef\\_vi\\_transmit\\_init\(\)](#) or [ef\\_vi\\_transmitv\\_init\(\)](#) before calling this function, and so in particular it is not legal to call this function more than once without initializing new descriptors in between those calls.

Definition at line 1437 of file ef\_vi.h.

### 11.11.2.20 ef\_vi\_transmitv

```
#define ef_vi_transmitv(  
    vi,  
    iov,  
    iov_len,  
    dma_id ) (vi)->ops.transmitv((vi), (iov), (iov_len), (dma_id))
```

Transmit a packet from a vector of packet buffers.

#### Parameters

<i>vi</i>	The virtual interface for which to initialize a TX descriptor.
<i>iov</i>	Start of the iovec array describing the packet buffers.
<i>iov_len</i>	Length of the iovec array.
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

#### Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

Transmit a packet from a vector of packet buffers. This initializes a TX descriptor on the TX descriptor ring, and submits it to the NIC. The NIC can then transmit a packet from the associated packet buffers.

This function simply wraps [ef\\_vi\\_transmitv\\_init\(\)](#) and [ef\\_vi\\_transmitv\\_push\(\)](#). It is provided as a convenience. It is less efficient than submitting the descriptors in batches by calling the functions separately, but unless there is a batch of packets to transmit, calling this function is often the right thing to do.

Building a packet by concatenating a vector of buffers allows:

- sending a packet that is larger than a packet buffer
  - the packet is split across multiple buffers in a vector
- optimizing sending packets with only small differences:
  - the packet is split into those parts that are constant, and those that vary between transmits
  - each part is written into its own buffer
  - after each transmit, the buffers containing varying data must be updated, but the buffers containing constant data are re-used
  - this minimizes the amount of data written between transmits.

Definition at line 1502 of file ef\_vi.h.

### 11.11.2.21 ef\_vi\_transmitv\_ctpio

```
#define ef_vi_transmitv_ctpio(  
    vi,  
    frame_len,  
    frame_iov,  
    frame_iov_len,  
    ct_threshold )
```

#### Value:

```
(vi)->ops.transmitv_ctpio((vi), (frame_len), (frame_iov), \  
    (frame_iov_len), (ct_threshold))
```

Transmit a packet using CTPIO from an array of buffers.

### Parameters

<i>vi</i>	The virtual interface on which to transmit.
<i>frame_len</i>	Frame length in bytes.
<i>frame_iov</i>	Buffers containing the frame to transmit.
<i>frame_iov_len</i>	Length of <i>frame_iov</i> .
<i>ct_threshold</i>	Number of bytes of the packet to buffer before starting to cut-through to the wire.

Transmit a packet using the CTPIO datapath. The CTPIO interface gives the lowest latency in most cases, and can be used by any number of VIs.

This function implements the latency critical part of a CTPIO send. It should be followed by a call to [ef\\_vi\\_transmit\\_ctpio\\_fallback\(\)](#) or similar before doing any further send calls on the same VI. [ef\\_vi\\_transmit\\_ctpio\\_fallback\(\)](#) provides a fallback frame which is sent in cases where the CTPIO send fails.

It is possible for a CTPIO send to fail for a number of reasons, including contention for adapter resources, and timeout due to the whole frame not being written to the adapter sufficiently quickly.

The *ct\_threshold* indicates how many bytes of the packet should be buffered by the adapter before starting to emit the packet. To disable cut-through behavior this must be at least as large as the frame length. To disable cut-through across all packet sizes, use `EF_VI_CTPIO_CT_THRESHOLD_SNF`.

The CTPIO path bypasses the adapter's normal transmit path, including checksum offloads, and so the packet is transmitted unmodified. The Ethernet FCS is appended as normal.

The caller must ensure that *frame\_len* is equal to the sum of the lengths in *frame\_iov*.

The buffers referenced by *frame\_iov* can be reused as soon as this call returns.

Definition at line 1907 of file `ef_vi.h`.

#### 11.11.2.22 ef\_vi\_transmitv\_ctpio\_copy

```
#define ef_vi_transmitv_ctpio_copy(  
    vi,  
    frame_len,  
    frame_iov,  
    frame_iov_len,  
    ct_threshold,  
    fallback )
```

#### Value:

```
(vi)->ops.transmitv_ctpio_copy((vi), (frame_len), (frame_iov), \  
    (frame_iov_len), (ct_threshold), \  
    (fallback))
```

Transmit a packet using CTPIO from an array of buffers, simultaneously copying the data into a fallback buffer.

### Parameters

<i>vi</i>	The virtual interface on which to transmit.
<i>frame_len</i>	Frame length in bytes.
<i>frame_iov</i>	Buffers containing the frame to transmit.
<i>frame_iov_len</i>	Length of <i>frame_iov</i> .
<i>ct_threshold</i>	Number of bytes of the packet to buffer before starting to cut-through to the wire.
<i>fallback</i>	Fallback buffer to copy the data into

This function is identical to `ef_vi_transmitv_cptio`, but additionally copies the data into a fallback buffer ready to provide to `ef_vi_transmit_ctpio_fallback`. This is an optimisation to avoid the need to copy the data in a separate step.

Definition at line 1929 of file `ef_vi.h`.

### 11.11.2.23 ef\_vi\_transmitv\_init

```
#define ef_vi_transmitv_init(  
    vi,  
    iov,  
    iov_len,  
    dma_id ) (vi)->ops.transmitv_init((vi), (iov), (iov_len), (dma_id))
```

Initialize TX descriptors on the TX descriptor ring, for a vector of packet buffers.

### Parameters

<i>vi</i>	The virtual interface for which to initialize a TX descriptor.
<i>iov</i>	Start of the iovec array describing the packet buffers.
<i>iov_len</i>	Length of the iovec array.
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

### Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

Initialize TX descriptors on the TX descriptor ring, for a vector of packet buffers. The associated packet buffers (identified in the *iov* vector) must contain the packet to transmit. This function only writes a few bytes into host memory, and is very fast.

Building a packet by concatenating a vector of buffers allows:

- sending a packet that is larger than a packet buffer
  - the packet is split across multiple buffers in a vector
- optimizing sending packets with only small differences:
  - the packet is split into those parts that are constant, and those that vary between transmits
  - each part is written into its own buffer
  - after each transmit, the buffers containing varying data must be updated, but the buffers containing constant data are re-used
  - this minimizes the amount of data written between transmits.

Definition at line 1419 of file `ef_vi.h`.

### 11.11.3 Typedef Documentation

#### 11.11.3.1 ef\_request\_id

```
typedef int ef_request_id
```

A DMA request identifier.

This is an integer token specified by the transport and associated with a DMA request. It is returned to the VI user with DMA completion events. It is typically used to identify the buffer associated with the transfer.

Definition at line 151 of file ef\_vi.h.

#### 11.11.3.2 ef\_vi

```
typedef struct ef_vi ef_vi
```

A virtual interface.

An [ef\\_vi](#) represents a virtual interface on a specific NIC. A virtual interface is a collection of an event queue and two DMA queues used to pass Ethernet frames between the transport implementation and the network.

Users should not access this structure.

### 11.11.4 Enumeration Type Documentation

#### 11.11.4.1 anonymous enum

```
anonymous enum
```

Possible types of events.

Enumerator

EF_EVENT_TYPE_RX	Good data was received.
EF_EVENT_TYPE_TX	Packets have been sent.
EF_EVENT_TYPE_RX_DISCARD	Data received and buffer consumed, but something is wrong.
EF_EVENT_TYPE_TX_ERROR	Transmit of packet failed.
EF_EVENT_TYPE_RX_NO_DESC_TRUNC	Received packet was truncated due to a lack of descriptors.
EF_EVENT_TYPE_SW	Software generated event.
EF_EVENT_TYPE_OFLOW	Event queue overflow.
EF_EVENT_TYPE_TX_WITH_TIMESTAMP	TX timestamp event.
EF_EVENT_TYPE_RX_PACKED_STREAM	A batch of packets was received in a packed stream.
EF_EVENT_TYPE_RX_MULTI	A batch of packets was received on a RX event merge vi.
EF_EVENT_TYPE_TX_ALT	Packet has been transmitted via a "TX alternative".
EF_EVENT_TYPE_RX_MULTI_DISCARD	A batch of packets was received with error condition set.

Definition at line 264 of file ef\_vi.h.

#### 11.11.4.2 anonymous enum

anonymous enum

The reason for an EF\_EVENT\_TYPE\_RX\_DISCARD event.

##### Enumerator

EF_EVENT_RX_DISCARD_CSUM_BAD	IP header or TCP/UDP checksum error
EF_EVENT_RX_DISCARD_MCAST_MISMATCH	Hash mismatch in a multicast packet
EF_EVENT_RX_DISCARD_CRC_BAD	Ethernet CRC error
EF_EVENT_RX_DISCARD_TRUNC	Frame was truncated
EF_EVENT_RX_DISCARD_RIGHTS	No ownership rights for the packet
EF_EVENT_RX_DISCARD_EV_ERROR	Event queue error, previous RX event has been lost
EF_EVENT_RX_DISCARD_OTHER	Other unspecified reason
EF_EVENT_RX_DISCARD_INNER_CSUM_BAD	Inner IP header or TCP/UDP checksum error
EF_EVENT_RX_DISCARD_MAX	Maximum value of this enumeration

Definition at line 360 of file ef\_vi.h.

#### 11.11.4.3 anonymous enum

anonymous enum

The reason for an EF\_EVENT\_TYPE\_TX\_ERROR event.

##### Enumerator

EF_EVENT_TX_ERROR_RIGHTS	No ownership rights for the packet
EF_EVENT_TX_ERROR_OFLOW	TX pacing engine work queue was full
EF_EVENT_TX_ERROR_2BIG	Oversized transfer has been indicated by the descriptor
EF_EVENT_TX_ERROR_BUS	Bus or descriptor protocol error occurred when attempting to read the memory referenced by the descriptor

Definition at line 415 of file ef\_vi.h.

#### 11.11.4.4 ef\_vi\_arch

enum `ef_vi_arch`

NIC architectures that are supported.

#### Enumerator

EF_VI_ARCH_FALCON	5000 and 6000-series NICs
EF_VI_ARCH_EF10	7000 and 8000-series NICs

Definition at line 578 of file ef\_vi.h.

#### 11.11.4.5 ef\_vi\_flags

enum `ef_vi_flags`

Flags that can be requested when allocating an `ef_vi`.

#### Enumerator

EF_VI_FLAGS_DEFAULT	Default setting
EF_VI_ISCSI_RX_HDIG	Receive iSCSI header digest enable: hardware verifies header digest (CRC) when packet is iSCSI
EF_VI_ISCSI_TX_HDIG	Transmit iSCSI header digest enable: hardware calculates and inserts header digest (CRC) when packet is iSCSI
EF_VI_ISCSI_RX_DDIG	Receive iSCSI data digest enable: hardware verifies data digest (CRC) when packet is iSCSI
EF_VI_ISCSI_TX_DDIG	Transmit iSCSI data digest enable: hardware calculates and inserts data digest (CRC) when packet is iSCSI
EF_VI_TX_PHYS_ADDR	Use physically addressed TX descriptor ring
EF_VI_RX_PHYS_ADDR	Use physically addressed RX descriptor ring
EF_VI_TX_IP_CSUM_DIS	IP checksum calculation and replacement is disabled
EF_VI_TX_TCPUDP_CSUM_DIS	TCP/UDP checksum calculation and replacement is disabled
EF_VI_TX_TCPUDP_ONLY	Drop transmit packets that are not TCP or UDP
EF_VI_TX_FILTER_IP	Drop packets with a mismatched IP source address (5000 and 6000 series only)
EF_VI_TX_FILTER_MAC	Drop packets with a mismatched MAC source address (5000 and 6000 series only)
EF_VI_TX_FILTER_MASK_1	Set lowest bit of queue ID to 0 when matching within filter block (5000 and 6000 series only)
EF_VI_TX_FILTER_MASK_2	Set lowest 2 bits of queue ID to 0 when matching within filter block (5000 and 6000 series only)
EF_VI_TX_FILTER_MASK_3	Set lowest 3 bits of queue ID to 0 when matching within filter block (5000 and 6000 series only)
EF_VI_TX_PUSH_DISABLE	Disable using TX descriptor push, so always use doorbell for transmit
EF_VI_TX_PUSH_ALWAYS	Always use TX descriptor push, so never use doorbell for transmit (7000 series and newer)
EF_VI_RX_TIMESTAMPS	Add timestamp to received packets (7000 series and newer)
EF_VI_TX_TIMESTAMPS	Add timestamp to transmitted packets (7000 series and newer), cannot be combined with EF_VI_TX_ALT
EF_VI_RX_PACKED_STREAM	Enable packed stream mode for received packets (7000 series and newer)

#### Enumerator

EF_VI_RX_PS_BUF_SIZE_64K	Use 64KiB packed stream buffers, instead of the 1024KiB default (7000 series and newer)
EF_VI_RX_EVENT_MERGE	Enable RX event merging mode for received packets; see <a href="#">ef_vi_receive_unbundle()</a> and <a href="#">ef_vi_receive_get_bytes()</a> for more details on using RX event merging mode
EF_VI_TX_ALT	Enable the "TX alternatives" feature (8000 series and newer), cannot be combined with EF_VI_TX_TIMESTAMPS
EF_VI_ENABLE_EV_TIMER	Controls whether the hardware event timer is enabled (8000 series and newer)
EF_VI_TX_CTPIO	Enable the "cut-through PIO" feature (X2000 series and newer).
EF_VI_TX_CTPIO_NO_POISON	When using CTPIO, prevent poisoned frames from reaching the wire (X2000 series and newer).

Definition at line 461 of file ef\_vi.h.

#### 11.11.4.6 ef\_vi\_layout\_type

enum [ef\\_vi\\_layout\\_type](#)

Types of layout that are used for receive buffers.

#### Enumerator

EF_VI_LAYOUT_FRAME	An Ethernet frameo
EF_VI_LAYOUT_MINOR_TICKS	Hardware timestamp (minor ticks) - 32 bits
EF_VI_LAYOUT_PACKET_LENGTH	Packet length - 16 bits

Definition at line 2137 of file ef\_vi.h.

#### 11.11.4.7 ef\_vi\_out\_flags

enum [ef\\_vi\\_out\\_flags](#)

Flags that can be returned when an [ef\\_vi](#) has been allocated.

#### Enumerator

EF_VI_OUT_CLOCK_SYNC_STATUS	Clock sync status
-----------------------------	-------------------

Definition at line 539 of file ef\_vi.h.

#### 11.11.4.8 ef\_vi\_rx\_discard\_err\_flags

enum [ef\\_vi\\_rx\\_discard\\_err\\_flags](#)

Flags that define which errors will cause RX\_DISCARD events.



### Enumerator

EF_VI_DISCARD_RX_L4_CSUM_ERR	TCP or UDP checksum error
EF_VI_DISCARD_RX_L3_CSUM_ERR	IP checksum error
EF_VI_DISCARD_RX_ETH_FCS_ERR	Ethernet FCS error
EF_VI_DISCARD_RX_ETH_LEN_ERR	Ethernet frame length error
EF_VI_DISCARD_RX_TOBE_DISC	To be discard in software (includes frame length error)

Definition at line 546 of file ef\_vi.h.

## 11.11.5 Function Documentation

### 11.11.5.1 ef\_eventq\_capacity()

```
int ef_eventq_capacity (  
    ef_vi * vi )
```

Returns the capacity of an event queue.

#### Parameters

<i>vi</i>	The event queue to query.
-----------	---------------------------

#### Returns

The capacity of an event queue.

Returns the capacity of an event queue. This is the maximum number of events that can be stored into the event queue before overflow.

It is up to the application to avoid event queue overflow by ensuring that the maximum number of events that can be delivered into an event queue is limited to its capacity. In general each RX descriptor and TX descriptor posted can cause an event to be generated.

In addition, when time-stamping is enabled time-sync events are generated at a rate of 4 per second. When TX timestamps are enabled you may get up to one event for each descriptor plus two further events per packet.

### 11.11.5.2 ef\_eventq\_current()

```
ef_vi_inline unsigned ef_eventq_current (  
    ef_vi * evq )
```

Get the current offset into the event queue.

#### Parameters

<i>evq</i>	The event queue to query.
------------	---------------------------

#### Returns

The current offset into the eventq.

Get the current offset into the event queue.

Definition at line 2126 of file ef\_vi.h.

#### 11.11.5.3 ef\_eventq\_has\_event()

```
int ef_eventq_has_event (
    const ef_vi * vi )
```

Returns true if [ef\\_eventq\\_poll\(\)](#) will return event(s)

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

True if [ef\\_eventq\\_poll\(\)](#) will return event(s).

Returns true if [ef\\_eventq\\_poll\(\)](#) will return event(s).

#### 11.11.5.4 ef\_eventq\_has\_many\_events()

```
int ef_eventq_has_many_events (
    const ef_vi * evq,
    int n_events )
```

Returns true if there are a given number of events in the event queue.

#### Parameters

<i>evq</i>	The event queue to query.
<i>n_events</i>	Number of events to check.

#### Returns

True if the event queue contains at least *n\_events* events.

Returns true if there are a given number of events in the event queue.

This looks ahead in the event queue, so has the property that it will not ping-pong a cache-line when it is called concurrently with events being delivered.

This function returns quickly. It is useful for an application to determine whether it is falling behind in its event processing.

#### 11.11.5.5 ef\_vi\_driver\_interface\_str()

```
const char* ef_vi_driver_interface_str (
    void )
```

Returns a string that identifies the char driver interface required.

##### Returns

A string that identifies the char driver interface required by this build of ef\_vi.

Returns a string that identifies the char driver interface required by this build of ef\_vi.

Returns the current version of the drivers that are running - useful to check that it is new enough.

#### 11.11.5.6 ef\_vi\_flags()

```
ef_vi_inline enum ef_vi_flags ef_vi_flags (
    ef_vi * vi )
```

Return the flags of the virtual interface.

##### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

##### Returns

The flags of the virtual interface.

Return the flags of the virtual interface.

Definition at line 917 of file ef\_vi.h.

#### 11.11.5.7 ef\_vi\_instance()

```
ef_vi_inline unsigned ef_vi_instance (
    ef_vi * vi )
```

Return the instance ID of the virtual interface.

##### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

##### Returns

The instance ID of the virtual interface.

Return the instance ID of the virtual interface.

Definition at line 932 of file ef\_vi.h.

#### 11.11.5.8 ef\_vi\_receive\_buffer\_len()

```
ef_vi_inline int ef_vi_receive_buffer_len (
    const ef_vi * vi )
```

Returns the length of a receive buffer.

### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

### Returns

The length of a receive buffer.

Returns the length of a receive buffer.

When a packet arrives that does not fit within a single receive buffer, it is spread over multiple buffers.

The application must ensure that receive buffers are at least as large as the value returned by this function, else there is a risk that a DMA may overrun the buffer.

Definition at line 1011 of file ef\_vi.h.

#### 11.11.5.9 ef\_vi\_receive\_capacity()

```
ef_vi_inline int ef_vi_receive_capacity (  
    const ef_vi * vi )
```

Returns the total capacity of the RX descriptor ring.

### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

### Returns

The total capacity of the RX descriptor ring, as slots for descriptor entries.

Returns the total capacity of the RX descriptor ring.

Definition at line 1080 of file ef\_vi.h.

#### 11.11.5.10 ef\_vi\_receive\_fill\_level()

```
ef_vi_inline int ef_vi_receive_fill_level (  
    const ef_vi * vi )
```

Returns the fill level of the RX descriptor ring.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The fill level of the RX descriptor ring, as slots for descriptor entries.

Returns the fill level of the RX descriptor ring. This is the number of slots that hold a descriptor (and an associated packet buffer). The fill level should be kept as high as possible, so there are enough slots available to handle a burst of incoming packets.

Definition at line 1064 of file ef\_vi.h.

#### 11.11.5.11 ef\_vi\_receive\_get\_bytes()

```
int ef_vi_receive_get_bytes (
    ef_vi * vi,
    const void * pkt,
    uint16_t * bytes_out )
```

Retrieve the number of bytes in a received packet in RX event merge mode.

#### Parameters

<i>vi</i>	The virtual interface that received the packet.
<i>pkt</i>	The first packet buffer for the received packet.
<i>bytes_out</i>	Pointer to a uint16_t, that is updated on return with the number of bytes in the packet.

Note that this function returns the number of bytes in a received packet, not received into a single buffer, ie it must only be called for the first buffer in a packet. For jumbos it will return the full length of the jumbo. Buffers prior to the last buffer in the packet will be filled completely.

The length does not include the length of the packet prefix.

#### Returns

0 on success, or a negative error code

#### 11.11.5.12 ef\_vi\_receive\_get\_timestamp()

```
int ef_vi_receive_get_timestamp (
    ef_vi * vi,
    const void * pkt,
    struct timespec * ts_out )
```

*Deprecated:* use [ef\\_vi\\_receive\\_get\\_timestamp\\_with\\_sync\\_flags\(\)](#) instead.

### Parameters

<i>vi</i>	The virtual interface that received the packet.
<i>pkt</i>	The received packet.
<i>ts_out</i>	Pointer to a timespec, that is updated on return with the UTC timestamp for the packet.

### Returns

0 on success, or a negative error code.

*This function is now deprecated. Use [ef\\_vi\\_receive\\_get\\_timestamp\\_with\\_sync\\_flags\(\)](#) instead.*

Retrieve the UTC timestamp associated with a received packet.

This function must be called after retrieving the associated RX event via [ef\\_eventq\\_poll\(\)](#), and before calling [ef\\_eventq\\_poll\(\)](#) again.

If the virtual interface does not have RX timestamps enabled, the behavior of this function is undefined.

#### 11.11.5.13 ef\_vi\_receive\_get\_timestamp\_with\_sync\_flags()

```
int ef_vi_receive_get_timestamp_with_sync_flags (
    ef_vi * vi,
    const void * pkt,
    struct timespec * ts_out,
    unsigned * flags_out )
```

Retrieve the UTC timestamp associated with a received packet, and the clock sync status flags.

### Parameters

<i>vi</i>	The virtual interface that received the packet.
<i>pkt</i>	The first packet buffer for the received packet.
<i>ts_out</i>	Pointer to a timespec, that is updated on return with the UTC timestamp for the packet.
<i>flags_out</i>	Pointer to an unsigned, that is updated on return with the sync flags for the packet.

### Returns

0 on success, or a negative error code:

- ENOMSG - Synchronization with adapter has not yet been achieved.  
This only happens with old firmware.
- ENODATA - Packet does not have a timestamp.  
On current Solarflare adapters, packets that are switched from TX to RX do not get timestamped.
- EL2NSYNC - Synchronization with adapter has been lost.  
This should never happen!

Retrieve the UTC timestamp associated with a received packet, and the clock sync status flags.

This function:

- must be called after retrieving the associated RX event via [ef\\_eventq\\_poll\(\)](#), and before calling [ef\\_eventq\\_poll\(\)](#) again
- must only be called for the first segment of a jumbo packet
- must not be called for any events other than RX.

If the virtual interface does not have RX timestamps enabled, the behavior of this function is undefined.

This function will also fail if the virtual interface has not yet synchronized with the adapter clock. This can take from a few hundred milliseconds up to several seconds from when the virtual interface is allocated.

On success the `ts_out` and `flags_out` fields are updated, and a value of zero is returned. The `flags_out` field contains the following flags:

- `EF_VI_SYNC_FLAG_CLOCK_SET` is set if the adapter clock has ever been set (in sync with system)
- `EF_VI_SYNC_FLAG_CLOCK_IN_SYNC` is set if the adapter clock is in sync with the external clock (PTP).

In case of error the timestamp result (`*ts_out`) is set to zero, and a non-zero error code is returned (see Return value above).

#### 11.11.5.14 ef\_vi\_receive\_post()

```
int ef_vi_receive_post (
    ef_vi * vi,
    ef_addr addr,
    ef_request_id dma_id )
```

Initialize an RX descriptor on the RX descriptor ring, and submit it to the NIC.

##### Parameters

<i>vi</i>	The virtual interface for which to initialize and push an RX descriptor.
<i>addr</i>	DMA address of the packet buffer to associate with the descriptor, as obtained from <a href="#">ef_memreg_dma_addr()</a> .
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

##### Returns

0 on success, or a negative error code.

Initialize an RX descriptor on the RX descriptor ring, and submit it to the NIC. The NIC can then receive a packet into the associated packet buffer.

This function simply wraps [ef\\_vi\\_receive\\_init\(\)](#) and [ef\\_vi\\_receive\\_push\(\)](#). It is provided as a convenience, but is less efficient than submitting the descriptors in batches by calling the functions separately.

Note that for Solarflare 7000-series NICs, this function submits RX descriptors only in multiples of 8. This is to conform with hardware requirements. If the number of newly initialized RX descriptors is not exactly divisible by 8, this function does not submit any remaining descriptors (including, potentially, the RX descriptor initialized in this call).

#### 11.11.5.15 ef\_vi\_receive\_prefix\_len()

```
ef_vi_inline int ef_vi_receive_prefix_len (
    const ef_vi * vi )
```

Returns the length of the prefix at the start of a received packet.

##### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

##### Returns

The length of the prefix at the start of a received packet.

Returns the length of the prefix at the start of a received packet.

The NIC may be configured to deliver meta-data in a prefix before the packet payload data. This call returns the size of the prefix.

When a large packet is received that is scattered over multiple packet buffers, the prefix is only present in the first buffer.

Definition at line 990 of file ef\_vi.h.

#### 11.11.5.16 ef\_vi\_receive\_query\_layout()

```
int ef_vi_receive_query_layout (
    ef_vi * vi,
    const ef_vi_layout_entry **const layout_out,
    int * layout_len_out )
```

Gets the layout of the data that the adapter delivers into receive buffers.

##### Parameters

<i>vi</i>	The virtual interface to query.
<i>layout_out</i>	Pointer to an ef_vi_layout_entry*, that is updated on return with a reference to the layout table.
<i>layout_len_out</i>	Pointer to an int, that is updated on return with the length of the layout table.

##### Returns

0 on success, or a negative error code.

Gets the layout of the data that the adapter delivers into receive buffers. Depending on the adapter type and options selected, there can be a meta-data prefix in front of each packet delivered into memory. Note that this prefix is per-packet, not per buffer, ie for jumbos the prefix will only be present in the first buffer of the packet.

The first entry is always of type EF\_VI\_LAYOUT\_FRAME, and the offset is the same as the value returned by [ef\\_vi\\_receive\\_prefix\\_len\(\)](#).

#### 11.11.5.17 ef\_vi\_receive\_set\_buffer\_len()

```
ef_vi_inline void ef_vi_receive_set_buffer_len (
    ef_vi * vi,
    unsigned buf_len )
```

Sets the length of receive buffers.



#### Parameters

<i>vi</i>	The virtual interface for which to set the length of receive buffers.
<i>buf_len</i>	The length of receive buffers.

Sets the length of receive buffers for this VI. The new length is used for subsequent calls to [ef\\_vi\\_receive\\_init\(\)](#) and [ef\\_vi\\_receive\\_post\(\)](#).

This call has no effect for 5000 and 6000-series (Falcon) adapters.

Definition at line 1028 of file ef\_vi.h.

#### 11.11.5.18 ef\_vi\_receive\_set\_discards()

```
int ef_vi_receive_set_discards (
    ef_vi * vi,
    unsigned discard_err_flags )
```

Set which errors cause an EF\_EVENT\_TYPE\_RX\_DISCARD event.

#### Parameters

<i>vi</i>	The virtual interface to configure.
<i>discard_err_flags</i>	Flags which indicate which errors will cause discard events

#### Returns

0 on success, or a negative error code.

Set which errors cause an EF\_EVENT\_TYPE\_RX\_DISCARD event

#### 11.11.5.19 ef\_vi\_receive\_space()

```
ef_vi_inline int ef_vi_receive_space (
    const ef_vi * vi )
```

Returns the amount of free space in the RX descriptor ring.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The amount of free space in the RX descriptor ring, as slots for descriptor entries.

Returns the amount of free space in the RX descriptor ring. This is the number of slots that are available for pushing a new descriptor (and an associated unfilled packet buffer).

Definition at line 1045 of file ef\_vi.h.

### 11.11.5.20 ef\_vi\_receive\_unbundle()

```
int ef_vi_receive_unbundle (
    ef_vi * ep,
    const ef_event * event,
    ef_request_id * ids )
```

Unbundle an event of type EF\_EVENT\_TYPE\_RX\_MULTI or EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD.

#### Parameters

<i>ep</i>	The virtual interface that has raised the event.
<i>event</i>	The event, of type EF_EVENT_TYPE_RX_MULTI or EF_EVENT_TYPE_RX_MULTI_DISCARD.
<i>ids</i>	Array of size EF_VI_RECEIVE_BATCH, that is updated on return with the DMA ids that were used in the original <a href="#">ef_vi_receive_init()</a> call.

#### Returns

The number of valid ef\_request\_ids (can be zero).

Unbundle an event of type EF\_EVENT\_TYPE\_RX\_MULTI or EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD.

In RX event merge mode the NIC will coalesce multiple packet receptions into a single RX event. This reduces PCIe load, enabling higher potential throughput at the cost of latency.

This function returns the number of descriptors whose reception has completed, and updates the ids array with the ef\_request\_ids for each completed DMA request.

After calling this function, the RX descriptors for the completed RX event are ready to be re-used.

In order to determine the length of each packet [ef\\_vi\\_receive\\_get\\_bytes\(\)](#) must be called, or the length examined in the packet prefix (see [ef\\_vi\\_receive\\_query\\_layout\(\)](#)).

### 11.11.5.21 ef\_vi\_resource\_id()

```
ef_vi_inline unsigned ef_vi_resource_id (
    ef_vi * vi )
```

Return the resource ID of the virtual interface.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The resource ID of the virtual interface.

Return the resource ID of the virtual interface.

Definition at line 903 of file ef\_vi.h.

### 11.11.5.22 ef\_vi\_set\_tx\_push\_threshold()

```
void ef_vi_set_tx_push_threshold (
    ef_vi * vi,
    unsigned threshold )
```

Set the threshold at which to switch from using TX descriptor push to using a doorbell.

#### Parameters

<i>vi</i>	The virtual interface for which to set the threshold.
<i>threshold</i>	The threshold to set, as the number of outstanding transmits at which to switch.

#### Returns

0 on success, or a negative error code.

Set the threshold at which to switch from using TX descriptor push to using a doorbell. TX descriptor push has better latency, but a doorbell is more efficient.

The default value for this is controlled using the EF\_VI\_TX\_PUSH\_DISABLE and EF\_VI\_TX\_PUSH\_ALWAYS flags to ef\_vi\_init().

This is not supported by all Solarflare NICs. At the time of writing, 7000-series NICs support this, but it is ignored by earlier NICs.

#### 11.11.5.23 ef\_vi\_transmit\_alt\_num\_ids()

```
unsigned ef_vi_transmit_alt_num_ids (  
    ef_vi * vi )
```

Return the number of TX alternatives allocated for a virtual interface.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The number of TX alternatives, or a negative error code.

Gets the number of TX alternatives for the given virtual interface.

#### 11.11.5.24 ef\_vi\_transmit\_alt\_query\_overhead()

```
int ef_vi_transmit_alt_query_overhead (  
    ef_vi * vi,  
    struct ef_vi_transmit_alt_overhead * params )
```

Query per-packet overhead parameters.

#### Parameters

<i>vi</i>	Interface to be queried
<i>params</i>	Returned overhead parameters

#### Returns

0 on success or -EINVAL if this VI doesn't support alternatives.

This function returns parameters which are needed by the ef\_vi\_transmit\_alt\_usage() function below.

### 11.11.5.25 ef\_vi\_transmit\_alt\_usage()

```
ef_vi_inline ef_vi_pure uint32_t ef_vi_transmit_alt_usage (
    const struct ef_vi_transmit_alt_overhead * params,
    uint32_t pkt_len )
```

Calculate a packet's buffer usage.

#### Parameters

<i>params</i>	Parameters returned by <code>ef_vi_transmit_alt_query_overhead</code>
<i>pkt_len</i>	Packet length in bytes

#### Returns

Packet buffer usage in bytes including per-packet overhead

This function calculates the number of bytes of buffering which will be used by the NIC to store a packet with the given length.

The returned value includes per-packet overheads, but does not include any other overhead which may be incurred by the hardware. Note that if the application has successfully requested N bytes of buffering using `ef_vi_transmit_alt_alloc()` then it is guaranteed to be able to store at least N bytes of packet data + per-packet overhead as calculated by this function.

It is possible that the application may be able to use more space in some situations if the non-per-packet overheads are low enough.

It is important that callers do not use `ef_vi_capabilities_get()` to query the available buffering. That function does not take into account non-per-packet overheads and so is likely to return more space than can actually be used by the application. This function is provided instead to allow applications to calculate their buffer usage accurately.

Definition at line 1837 of file `ef_vi.h`.

### 11.11.5.26 ef\_vi\_transmit\_capacity()

```
ef_vi_inline int ef_vi_transmit_capacity (
    const ef_vi * vi )
```

Returns the total capacity of the TX descriptor ring.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The total capacity of the TX descriptor ring, as slots for descriptor entries.

Returns the total capacity of the TX descriptor ring.

Definition at line 1359 of file `ef_vi.h`.

### 11.11.5.27 ef\_vi\_transmit\_ctpio()

```
ef_vi_inline void ef_vi_transmit_ctpio (  
    ef_vi * vi,  
    const void * frame_buf,  
    size_t frame_len,  
    unsigned ct_threshold )
```

Transmit a packet using CTPIO.

#### Parameters

<i>vi</i>	The virtual interface on which to transmit.
<i>frame_buf</i>	Buffer containing the frame to transmit.
<i>frame_len</i>	Frame length in bytes.
<i>ct_threshold</i>	Number of bytes of the packet to buffer before starting to cut-through to the wire.

Transmit a packet using the CTPIO datapath. The CTPIO interface gives the lowest latency in most cases, and can be used by any number of VIs.

This function implements the latency critical part of a CTPIO send. It should be followed by a call to [ef\\_vi\\_transmit\\_ctpio\\_fallback\(\)](#) or similar before doing any further send calls on the same VI. [ef\\_vi\\_transmit\\_ctpio\\_fallback\(\)](#) provides a fallback frame which is sent in cases where the CTPIO send fails.

It is possible for a CTPIO send to fail for a number of reasons, including contention for adapter resources, and timeout due to the whole frame not being written to the adapter sufficiently quickly.

The *ct\_threshold* indicates how many bytes of the packet should be buffered by the adapter before starting to emit the packet. To disable cut-through behavior this must be at least as large as the frame length. To disable cut-through across all packet sizes, use `EF_VI_CTPIO_CT_THRESHOLD_SNF`.

The CTPIO path bypasses the adapter's normal transmit path, including checksum offloads, and so the packet is transmitted unmodified. The Ethernet FCS is appended as normal.

The buffer *frame\_buf* can be reused as soon as this call returns.

Definition at line 1970 of file *ef\_vi.h*.

### 11.11.5.28 ef\_vi\_transmit\_ctpio\_fallback()

```
int ef_vi_transmit_ctpio_fallback (  
    ef_vi * vi,  
    ef_addr dma_addr,  
    size_t len,  
    ef_request_id dma_id )
```

Post fallback frame for a CTPIO transmit.

#### Parameters

<i>vi</i>	The virtual interface on which to transmit.
<i>dma_addr</i>	DMA address of frame.
<i>len</i>	Frame length in bytes.
<i>dma_id</i>	DMA ID to be returned on completion.

## Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

This should be called after `ef_vi_transmit_ctpio()` or similar. It provides the fallback frame which is sent in the event that a CTPIO transmit fails.

Ordinarily the fallback frame will be the same as the CTPIO frame, but it doesn't have to be.

### 11.11.5.29 ef\_vi\_transmit\_fill\_level()

```
ef_vi_inline int ef_vi_transmit_fill_level (
    const ef_vi * vi )
```

Returns the fill level of the TX descriptor ring.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

## Returns

The fill level of the TX descriptor ring, as slots for descriptor entries.

Returns the fill level of the TX descriptor ring. This is the number of slots that hold a descriptor (and an associated filled packet buffer). The fill level should be low or 0, unless a large number of packets have recently been posted for transmission. A consistently high fill level should be investigated.

Definition at line 1343 of file `ef_vi.h`.

### 11.11.5.30 ef\_vi\_transmit\_init()

```
int ef_vi_transmit_init (
    ef_vi * vi,
    ef_addr addr,
    int bytes,
    ef_request_id dma_id )
```

Initialize a TX descriptor on the TX descriptor ring, for a single packet buffer.

#### Parameters

<i>vi</i>	The virtual interface for which to initialize a TX descriptor.
<i>addr</i>	DMA address of the packet buffer to associate with the descriptor, as obtained from <code>ef_memreg_dma_addr()</code> .
<i>bytes</i>	The size of the packet to transmit.
<i>dma_id</i>	DMA id to associate with the descriptor. This is completely arbitrary, and can be used for subsequent tracking of buffers.

## Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

Initialize a TX descriptor on the TX descriptor ring, for a single packet buffer. The associated packet buffer (identified by its DMA address) must contain the packet to transmit. This function only writes a few bytes into host memory, and is very fast.

### 11.11.5.31 ef\_vi\_transmit\_init\_undo()

```
void ef_vi_transmit_init_undo (
    ef_vi * vi )
```

Remove all TX descriptors from the TX descriptor ring that have been initialized since last transmit.

#### Parameters

<i>vi</i>	The virtual interface for which to remove initialized TX descriptors from the TX descriptor ring.
-----------	---

#### Returns

None

Remove all TX descriptors from the TX descriptor ring that have been initialized since last transmit. This will undo the effects of calls made to `ef_vi_transmit_init` or `ef_vi_transmitv_init` since the last "push".

Initializing and then removing descriptors can have a warming effect on the transmit code path for subsequent transmits. This can reduce latency jitter caused by code and state being evicted from cache during delays between transmitting packets. This technique is also effective before the first transmit.

### 11.11.5.32 ef\_vi\_transmit\_space()

```
ef_vi_inline int ef_vi_transmit_space (
    const ef_vi * vi )
```

Returns the amount of free space in the TX descriptor ring.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The amount of free space in the TX descriptor ring, as slots for descriptor entries.

Returns the amount of free space in the TX descriptor ring. This is the number of slots that are available for pushing a new descriptor (and an associated filled packet buffer).

Definition at line 1323 of file `ef_vi.h`.

### 11.11.5.33 ef\_vi\_transmit\_unbundle()

```
int ef_vi_transmit_unbundle (
    ef_vi * ep,
    const ef_event * event,
    ef_request_id * ids )
```

Unbundle an event of type of type `EF_EVENT_TYPE_TX` or `EF_EVENT_TYPE_TX_ERROR`.

### Parameters

<i>ep</i>	The virtual interface that has raised the event.
<i>event</i>	The event, of type EF_EVENT_TYPE_TX or EF_EVENT_TYPE_TX_ERROR
<i>ids</i>	Array of size EF_VI_TRANSMIT_BATCH, that is updated on return with the DMA ids that were used in the originating ef_vi_transmit_*() calls.

### Returns

The number of valid ef\_request\_ids (can be zero).

Unbundle an event of type of type EF\_EVENT\_TYPE\_TX or EF\_EVENT\_TYPE\_TX\_ERROR.

The NIC might coalesce multiple packet transmissions into a single TX event in the event queue. This function returns the number of descriptors whose transmission has completed, and updates the ids array with the ef\_request\_ids for each completed DMA request.

After calling this function, the TX descriptors for the completed TX event are ready to be re-initialized. The associated packet buffers are no longer in use by ef\_vi. Each buffer can then be freed, or can be re-used (for example as a packet buffer for a descriptor on the TX ring, or on the RX ring).

#### 11.11.5.34 ef\_vi\_transmitv\_ctpio\_fallback()

```
int ef_vi_transmitv_ctpio_fallback (
    ef_vi * vi,
    const ef_iovec * dma_iov,
    int dma_iov_len,
    ef_request_id dma_id )
```

Post fallback frame for a CTPIO transmit.

### Parameters

<i>vi</i>	The virtual interface on which to transmit.
<i>dma_iov</i>	Array of source buffer DMA addresses.
<i>dma_iov_len</i>	Length of dma_iov.
<i>dma_id</i>	DMA ID to be returned on completion.

### Returns

0 on success, or a negative error code:  
-EAGAIN if the descriptor ring is full.

This should be called after ef\_vi\_transmit\_ctpio() or similar. It provides the fallback frame which is sent in the event that a CTPIO transmit fails.

Ordinarily the fallback frame will be the same as the CTPIO frame, but it doesn't have to be.



### 11.11.5.35 ef\_vi\_version\_str()

```
const char* ef_vi_version_str (  
    void )
```

Return a string that identifies the version of [ef\\_vi](#).

#### Returns

A string that identifies the version of [ef\\_vi](#).

Return a string that identifies the version of [ef\\_vi](#). This should be treated as an unstructured string. At time of writing it is the version of OpenOnload or EnterpriseOnload in which [ef\\_vi](#) is distributed.

Note that Onload will check this is a version that it recognizes. It recognizes the version strings generated by itself, and those generated by older official releases of Onload (when the API hasn't changed), but not those generated by older patched releases of Onload. Consequently, [ef\\_vi](#) applications built against patched versions of Onload will not be supported by future versions of Onload.

## 11.12 memreg.h File Reference

Registering memory for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/base.h>
```

### Data Structures

- struct [ef\\_memreg](#)  
*Memory that has been registered for use with [ef\\_vi](#).*

### Typedefs

- typedef struct [ef\\_memreg](#) [ef\\_memreg](#)  
*Memory that has been registered for use with [ef\\_vi](#).*

### Functions

- int [ef\\_memreg\\_alloc](#) ([ef\\_memreg](#) \*mr, [ef\\_driver\\_handle](#) mr\_dh, struct [ef\\_pd](#) \*pd, [ef\\_driver\\_handle](#) pd\_dh, void \*p\_mem, size\_t len\_bytes)  
*Register a memory region for use with [ef\\_vi](#).*
- int [ef\\_memreg\\_free](#) ([ef\\_memreg](#) \*mr, [ef\\_driver\\_handle](#) mr\_dh)  
*Unregister a memory region.*
- [ef\\_vi\\_inline](#) [ef\\_addr](#) [ef\\_memreg\\_dma\\_addr](#) ([ef\\_memreg](#) \*mr, size\_t offset)  
*Return the DMA address for the given offset within a registered memory region.*

## 11.12.1 Detailed Description

Registering memory for EtherFabric Virtual Interface HAL.

### Author

Solarflare Communications, Inc.

### Date

2018/11/06

### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

## 11.12.2 Function Documentation

### 11.12.2.1 ef\_memreg\_alloc()

```
int ef_memreg_alloc (
    ef_memreg * mr,
    ef_driver_handle mr_dh,
    struct ef_pd * pd,
    ef_driver_handle pd_dh,
    void * p_mem,
    size_t len_bytes )
```

Register a memory region for use with [ef\\_vi](#).

#### Parameters

<i>mr</i>	The <a href="#">ef_memreg</a> object to initialize.
<i>mr_dh</i>	Driver handle for the <a href="#">ef_memreg</a> .
<i>pd</i>	Protection domain in which to register memory.
<i>pd_dh</i>	Driver handle for the protection domain.
<i>p_mem</i>	Start of memory region to be registered. This must be page-aligned, and so be on a 4K boundary.
<i>len_bytes</i>	Length of memory region to be registered. This must be a multiple of the packet buffer size (currently 2048 bytes).

## Returns

0 on success, or a negative error code.

Register memory for use with [ef\\_vi](#).

Before calling this function, the memory must be allocated. using malloc or similar.

After calling this function, the memory is registered, and can be used for DMA buffers. [ef\\_memreg\\_dma\\_addr\(\)](#) can then be used to obtain DMA addresses for buffers within the registered area.

Registered memory is associated with a particular protection domain, and the DMA addresses can be used only with virtual interfaces that are associated with the same protection domain. Memory can be registered with multiple protection domains so that a single pool of buffers can be used with multiple virtual interfaces.

Memory that is registered is pinned, and therefore it cannot be swapped out to disk.

## Note

If an application that has registered memory forks, then copy-on-write semantics can cause new pages to be allocated which are not registered. This problem can be solved either by ensuring that the registered memory regions are shared by parent and child (e.g. by using MAP\_SHARED), or by using `madvise(MADV_DONTFORK)` to prevent the registered memory from being accessible in the child.

### 11.12.2.2 ef\_memreg\_dma\_addr()

```
ef_vi_inline ef_addr ef_memreg_dma_addr (
    ef_memreg * mr,
    size_t offset )
```

Return the DMA address for the given offset within a registered memory region.

#### Parameters

<i>mr</i>	The <a href="#">ef_memreg</a> object to query.
<i>offset</i>	The offset within the <a href="#">ef_memreg</a> object.

## Returns

The DMA address for the given offset within a registered memory region.

Return the DMA address for the given offset within a registered memory region.

Note that DMA addresses are only contiguous within each 4K block of a memory region.

Definition at line 132 of file memreg.h.

### 11.12.2.3 ef\_memreg\_free()

```
int ef_memreg_free (
    ef_memreg * mr,
    ef_driver_handle mr_dh )
```

Unregister a memory region.

### Parameters

<i>mr</i>	The <a href="#">ef_memreg</a> object to unregister.
<i>mr_dh</i>	Driver handle for the <a href="#">ef_memreg</a> .

### Returns

0 on success, or a negative error code.

Unregister a memory region.

### Note

To free all the resources, the driver handle associated with the memory must also be closed by calling [ef\\_driver\\_close\(\)](#).

## 11.13 packedstream.h File Reference

Packed streams for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/base.h>
```

### Data Structures

- struct [ef\\_packed\\_stream\\_packet](#)  
*Per-packet meta-data.*
- struct [ef\\_packed\\_stream\\_params](#)  
*Packed-stream mode parameters.*

### Macros

- #define [EF\\_VI\\_PS\\_FLAG\\_CLOCK\\_SET](#) 0x1  
*Set if the adapter clock has ever been set (in sync with system).*
- #define [EF\\_VI\\_PS\\_FLAG\\_CLOCK\\_IN\\_SYNC](#) 0x2  
*Set if the adapter clock is in sync with the external clock (PTP).*
- #define [EF\\_VI\\_PS\\_FLAG\\_BAD\\_FCS](#) 0x4  
*Set if a bad Frame Check Sequence has occurred.*
- #define [EF\\_VI\\_PS\\_FLAG\\_BAD\\_L4\\_CSUM](#) 0x8  
*Set if a layer-4 (TCP/UDP) checksum error is detected, or if a good layer-4 checksum is not detected (depending on adapter).*
- #define [EF\\_VI\\_PS\\_FLAG\\_BAD\\_L3\\_CSUM](#) 0x10  
*Set if a layer-3 (IPv4) checksum error is detected, or if a good layer-3 checksum is not detected (depending on adapter).*
- #define [EF\\_VI\\_PS\\_FLAG\\_BAD\\_IP\\_CSUM](#)  
*Retained for backwards compatibility. Do not use.*

## Functions

- int `ef_vi_packed_stream_get_params` (`ef_vi` \*vi, `ef_packed_stream_params` \*psp\_out)  
*Get the parameters for packed-stream mode.*
- int `ef_vi_packed_stream_unbundle` (`ef_vi` \*vi, const `ef_event` \*ev, `ef_packed_stream_packet` \*\*pkt\_iter, int \*n\_pkts\_out, int \*n\_bytes\_out)  
*Unbundle an event of type EF\_EVENT\_TYPE\_RX\_PACKED\_STREAM.*
- static `ef_packed_stream_packet` \* `ef_packed_stream_packet_first` (void \*start\_of\_buffer, int psp\_start\_offset)  
*Get the metadata for the first packet in a packed stream.*
- static `ef_packed_stream_packet` \* `ef_packed_stream_packet_next` (`ef_packed_stream_packet` \*ps\_pkt)  
*Get the metadata for the next packet in a packed stream.*
- static void \* `ef_packed_stream_packet_payload` (`ef_packed_stream_packet` \*ps\_pkt)  
*Return a pointer to the packet payload.*

### 11.13.1 Detailed Description

Packed streams for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.13.2 Macro Definition Documentation

#### 11.13.2.1 EF\_VI\_PS\_FLAG\_BAD\_IP\_CSUM

```
#define EF_VI_PS_FLAG_BAD_IP_CSUM
```

#### Value:

```
(EF_VI_PS_FLAG_BAD_L4_CSUM | \ EF_VI_PS_FLAG_BAD_L3_CSUM)
```

Retained for backwards compatibility. Do not use.

Definition at line 83 of file packedstream.h.

## 11.13.3 Function Documentation

### 11.13.3.1 ef\_packed\_stream\_packet\_first()

```
static ef_packed_stream_packet* ef_packed_stream_packet_first (
    void * start_of_buffer,
    int psp_start_offset ) [inline], [static]
```

Get the metadata for the first packet in a packed stream.

#### Parameters

<i>start_of_buffer</i>	Pointer to the start of the buffer.
<i>psp_start_offset</i>	Offset within the buffer to the start of the packet.

#### Returns

Pointer to packet metadata

Get the metadata for the first packet in a packed stream.

The packet is identified by its storage location, as an offset within a buffer.

Definition at line 179 of file packedstream.h.

### 11.13.3.2 ef\_packed\_stream\_packet\_next()

```
static ef_packed_stream_packet* ef_packed_stream_packet_next (
    ef_packed_stream_packet * ps_pkt ) [inline], [static]
```

Get the metadata for the next packet in a packed stream.

#### Parameters

<i>ps_pkt</i>	Pointer to a packed stream packet.
---------------	------------------------------------

#### Returns

Pointer to the next packed stream packet.

Get the metadata for the next packet in a packed stream.

The packet is identified by giving the current packet in the iteration.

Definition at line 196 of file packedstream.h.

### 11.13.3.3 ef\_packed\_stream\_packet\_payload()

```
static void* ef_packed_stream_packet_payload (
    ef_packed_stream_packet * ps_pkt ) [inline], [static]
```

Return a pointer to the packet payload.

#### Parameters

<i>ps_pkt</i>	Pointer to a packed stream packet.
---------------	------------------------------------

#### Returns

Pointer to the packet payload.

Return a pointer to the packet payload.

Definition at line 211 of file packedstream.h.

#### 11.13.3.4 ef\_vi\_packed\_stream\_get\_params()

```
int ef_vi_packed_stream_get_params (
    ef_vi * vi,
    ef_packed_stream_params * psp_out )
```

Get the parameters for packed-stream mode.

#### Parameters

<i>vi</i>	The virtual interface to query.
<i>psp_out</i>	Pointer to an <a href="#">ef_packed_stream_params</a> , that is updated on return with the parameters for packed-stream mode.

#### Returns

0 on success, or a negative error code:  
-EINVAL if the virtual interface is not in packed-stream mode.

Get the parameters for packed-stream mode.

#### 11.13.3.5 ef\_vi\_packed\_stream\_unbundle()

```
int ef_vi_packed_stream_unbundle (
    ef_vi * vi,
    const ef_event * ev,
    ef_packed_stream_packet ** pkt_iter,
    int * n_pkts_out,
    int * n_bytes_out )
```

Unbundle an event of type EF\_EVENT\_TYPE\_RX\_PACKED\_STREAM.

#### Parameters

<i>vi</i>	The virtual interface that has raised the event.
<i>ev</i>	The event, of type EF_EVENT_TYPE_RX_PACKED_STREAM.
<i>pkt_iter</i>	Pointer to an <a href="#">ef_packed_stream_packet*</a> , that is updated on return with the value for the next call to this function. See below for more details.
<i>n_pkts_out</i>	Pointer to an int, that is updated on return with the number of packets unpacked.
<i>n_bytes_out</i>	Pointer to an int, that is updated on return with the number of bytes unpacked.

## Returns

0 on success, or a negative error code.

Unbundle an event of type `EF_EVENT_TYPE_RX_PACKED_STREAM`.

This function should be called once for each `EF_EVENT_TYPE_RX_PACKED_STREAM` event received.

If `EF_EVENT_RX_PS_NEXT_BUFFER(*ev)` is true, `*pkt_iter` should be initialized to the value returned by `ef_packed_stream_packet_first()`.

When `EF_EVENT_RX_PS_NEXT_BUFFER(*ev)` is not true, `*pkt_iter` should contain the value left by the previous call. After each call `*pkt_iter` points at the location where the next packet will be delivered.

The return value is 0, or a negative error code. If the error code is `-ENOMSG`, `-ENODATA` or `-EL2NSYNC` then there was a problem with the hardware timestamp: see `ef_vi_receive_get_timestamp_with_sync_flags()` for details.

## 11.14 pd.h File Reference

Protection Domains for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/base.h>
```

### Data Structures

- struct `ef_pd`  
*A protection domain.*

### Macros

- `#define EF_PD_VLAN_NONE -1`  
*May be passed to `ef_pd_alloc_with_vport()` to indicate that the PD is not associated with a particular VLAN.*

### Typedefs

- typedef struct `ef_pd` `ef_pd`  
*A protection domain.*

### Enumerations

- enum `ef_pd_flags` {  
`EF_PD_DEFAULT` = 0x0, `EF_PD_VF` = 0x1, `EF_PD_PHYS_MODE` = 0x2, `EF_PD_RX_PACKED_STREAM` = 0x4,  
`EF_PD_VPORT` = 0x8, `EF_PD_MCAST_LOOP` = 0x10, `EF_PD_MEMREG_64KiB` = 0x20 }  
*Flags for a protection domain.*



## Functions

- int `ef_pd_alloc` (`ef_pd *pd`, `ef_driver_handle pd_dh`, int ifindex, enum `ef_pd_flags` flags)  
*Allocate a protection domain.*
- int `ef_pd_alloc_by_name` (`ef_pd *pd`, `ef_driver_handle pd_dh`, const char \*cluster\_or\_intf\_name, enum `ef_pd_flags` flags)  
*Allocate a protection domain for a named interface or cluster.*
- int `ef_pd_alloc_with_vport` (`ef_pd *pd`, `ef_driver_handle pd_dh`, const char \*intf\_name, enum `ef_pd_flags` flags, int vlan\_id)  
*Allocate a protection domain with vport support.*
- const char \* `ef_pd_interface_name` (`ef_pd *pd`)  
*Look up the interface being used by the protection domain.*
- int `ef_pd_free` (`ef_pd *pd`, `ef_driver_handle pd_dh`)  
*Free a protection domain.*

### 11.14.1 Detailed Description

Protection Domains for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.14.2 Enumeration Type Documentation

#### 11.14.2.1 `ef_pd_flags`

enum `ef_pd_flags`

Flags for a protection domain.

#### Enumerator

<code>EF_PD_DEFAULT</code>	Default flags
<code>EF_PD_VF</code>	Protection domain uses a virtual function and the system IOMMU instead of NIC buffer table.
<code>EF_PD_PHYS_MODE</code>	Protection domain uses physical addressing mode
<code>EF_PD_RX_PACKED_STREAM</code>	Protection domain supports packed stream mode
<code>EF_PD_VPORT</code>	Protection domain supports virtual ports
<code>EF_PD_MCAST_LOOP</code>	Protection domain supports HW multicast loopback
<code>EF_PD_MEMREG_64KiB</code>	Protection domain uses $\geq$ 64KB registered memory mappings

Definition at line 50 of file pd.h.

## 11.14.3 Function Documentation

### 11.14.3.1 ef\_pd\_alloc()

```
int ef_pd_alloc (
    ef_pd * pd,
    ef_driver_handle pd_dh,
    int ifindex,
    enum ef_pd_flags flags )
```

Allocate a protection domain.

#### Parameters

<i>pd</i>	Memory to use for the allocated protection domain.
<i>pd_dh</i>	The <code>ef_driver_handle</code> to associate with the protection domain.
<i>ifindex</i>	Index of the interface to use for the protection domain.
<i>flags</i>	Flags to specify protection domain properties.

#### Returns

0 on success, or a negative error code.

Allocate a protection domain.

Allocates a 'protection domain' which specifies how memory should be protected for your VIs. For supported modes - see [Packet Buffer Addressing](#)

#### Note

If you are using a 'hardened' kernel (e.g. Gentoo-hardened) then this is the first call which will probably fail. Currently, the only workaround to this is to run as root.

Use "if\_nametoindex" to find the index of an interface, which needs to be the physical interface (i.e. eth2, not eth2.6 or bond0 or similar.)

### 11.14.3.2 ef\_pd\_alloc\_by\_name()

```
int ef_pd_alloc_by_name (
    ef_pd * pd,
    ef_driver_handle pd_dh,
    const char * cluster_or_intf_name,
    enum ef_pd_flags flags )
```

Allocate a protection domain for a named interface or cluster.

#### Parameters

<i>pd</i>	Memory to use for the allocated protection domain.
<i>pd_dh</i>	An <code>ef_driver_handle</code> .
<i>cluster_or_intf_name</i>	Name of cluster, or name of interface.
<i>flags</i>	Flags to specify protection domain properties.

#### Returns

0 on success, or a negative error code.

Allocate a protection domain, trying first from a cluster of the given name, or if no cluster of that name exists assume that `cluster_or_intf_name` is the name of an interface.

When `cluster_or_intf_name` gives the name of a cluster it may optionally be prefixed with a channel number. For example: "0@cluster". In this case the specified channel instance within the cluster is allocated.

#### 11.14.3.3 ef\_pd\_alloc\_with\_vport()

```
int ef_pd_alloc_with_vport (
    ef_pd * pd,
    ef_driver_handle pd_dh,
    const char * intf_name,
    enum ef_pd_flags flags,
    int vlan_id )
```

Allocate a protection domain with vport support.

#### Parameters

<i>pd</i>	Memory to use for the allocated protection domain.
<i>pd_dh</i>	The <code>ef_driver_handle</code> to associate with the protection domain.
<i>intf_name</i>	Name of interface to use for the protection domain.
<i>flags</i>	Flags to specify protection domain properties.
<i>vlan_id</i>	The vlan id to associate with the protection domain.

#### Returns

0 on success, or a negative error code.

Allocate a protection domain with vport support.

Solarflare adapters have an internal switch to connect virtual ports (vports) to functions. This call is used to add an extra vport to a function, typically so that the function can then pass traffic to itself between an existing vport and the extra vport.

The `vlan_id` can be either `EF_PD_VLAN_NONE`, or the id of a vlan to associate a vlan with the vport.

This call requires full-featured firmware.

#### 11.14.3.4 ef\_pd\_free()

```
int ef_pd_free (
    ef_pd * pd,
    ef_driver_handle pd_dh )
```

Free a protection domain.

#### Parameters

<i>pd</i>	Memory used by the protection domain.
<i>pd_dh</i>	The ef_driver_handle associated with the protection domain.

#### Returns

0 on success, or a negative error code.

Free a protection domain.

To free up all resources, you must also close the associated driver handle.

You should call this when you're finished; although they will be cleaned up when the application exits, if you don't.

Be very sure that you don't try and re-use the vi/pd/driver structure after it has been freed.

#### 11.14.3.5 ef\_pd\_interface\_name()

```
const char* ef_pd_interface_name (
    ef_pd * pd )
```

Look up the interface being used by the protection domain.

#### Parameters

<i>pd</i>	Memory used by the protection domain.
-----------	---------------------------------------

#### Returns

The interface being used by the protection domain. NULL when interface was not accessible for user context.

Look up the interface being used by the protection domain.

## 11.15 pio.h File Reference

Programmed Input/Output for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/base.h>
```

#### Data Structures

- struct `ef_pio`  
*A Programmed I/O region.*

## Macros

- #define `EF_VI_CONFIG_PIO` 1  
*True if Programmed I/O regions can be configured.*

## Typedefs

- typedef struct `ef_pio` `ef_pio`  
*A Programmed I/O region.*

## Functions

- int `ef_pio_alloc` (`ef_pio` \*pio, `ef_driver_handle` pio\_dh, struct `ef_pd` \*pd, unsigned len\_hint, `ef_driver_handle` pd\_dh)  
*Allocate a Programmed I/O region.*
- int `ef_vi_get_pio_size` (`ef_vi` \*vi)  
*Get the size of the Programmed I/O region.*
- int `ef_pio_free` (`ef_pio` \*pio, `ef_driver_handle` pio\_dh)  
*Free a Programmed I/O region.*
- int `ef_pio_link_vi` (`ef_pio` \*pio, `ef_driver_handle` pio\_dh, struct `ef_vi` \*vi, `ef_driver_handle` vi\_dh)  
*Link a Programmed I/O region with a virtual interface.*
- int `ef_pio_unlink_vi` (`ef_pio` \*pio, `ef_driver_handle` pio\_dh, struct `ef_vi` \*vi, `ef_driver_handle` vi\_dh)  
*Unlink a Programmed I/O region from a virtual interface.*
- int `ef_pio_memcpy` (`ef_vi` \*vi, const void \*base, int offset, int len)  
*Copy data from memory into a Programmed I/O region.*

### 11.15.1 Detailed Description

Programmed Input/Output for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.15.2 Function Documentation

### 11.15.2.1 ef\_pio\_alloc()

```
int ef_pio_alloc (
    ef_pio * pio,
    ef_driver_handle pio_dh,
    struct ef_pd * pd,
    unsigned len_hint,
    ef_driver_handle pd_dh )
```

Allocate a Programmed I/O region.

#### Parameters

<i>pio</i>	Memory to use for the allocated Programmed I/O region.
<i>pio_dh</i>	The <code>ef_driver_handle</code> to associate with the Programmed I/O region.
<i>pd</i>	The protection domain to associate with the Programmed I/O region.
<i>len_hint</i>	Hint for the requested length of the Programmed I/O region.
<i>pd_dh</i>	The <code>ef_driver_handle</code> for the protection domain.

#### Returns

0 on success, or a negative error code.

Allocate a Programmed I/O region.

This function is available only on 64-bit x86 processors.

#### 11.15.2.2 ef\_pio\_free()

```
int ef_pio_free (
    ef_pio * pio,
    ef_driver_handle pio_dh )
```

Free a Programmed I/O region.

#### Parameters

<i>pio</i>	The Programmed I/O region.
<i>pio_dh</i>	The <code>ef_driver_handle</code> for the Programmed I/O region.

#### Returns

0 on success, or a negative error code.

Free a Programmed I/O region.

The Programmed I/O region must not be linked when this function is called. See [ef\\_pio\\_unlink\\_vi\(\)](#).

To free up all resources, the associated driver handle must then be closed by calling [ef\\_driver\\_close\(\)](#).

#### 11.15.2.3 ef\_pio\_link\_vi()

```
int ef_pio_link_vi (
    ef_pio * pio,
    ef_driver_handle pio_dh,
    struct ef_vi * vi,
    ef_driver_handle vi_dh )
```

Link a Programmed I/O region with a virtual interface.

### Parameters

<i>pio</i>	The Programmed I/O region.
<i>pio_dh</i>	The ef_driver_handle for the Programmed I/O region.
<i>vi</i>	The virtual interface to link with the Programmed I/O region.
<i>vi_dh</i>	The ef_driver_handle for the virtual interface.

### Returns

0 on success, or a negative error code.

Link a Programmed I/O region with a virtual interface. Only one region can be linked to a virtual interface.

#### 11.15.2.4 ef\_pio\_memcpy()

```
int ef_pio_memcpy (
    ef_vi * vi,
    const void * base,
    int offset,
    int len )
```

Copy data from memory into a Programmed I/O region.

### Parameters

<i>vi</i>	The virtual interface for the Programmed I/O region.
<i>base</i>	The base address of the memory to copy.
<i>offset</i>	The offset into the Programmed I/O region at which to copy the data to. You shouldn't try to copy memory to part of the PIO region that is already in use for an ongoing send as this may result in corruption.
<i>len</i>	The number of bytes to copy.

### Returns

0 on success, or a negative error code.

This function copies data from the user buffer to the adapter's PIO buffer. It goes via an intermediate buffer to meet any alignment requirements that the adapter may have.

Please refer to the PIO transmit functions (e.g.: [ef\\_vi\\_transmit\\_pio\(\)](#) and [ef\\_vi\\_transmit\\_copy\\_pio\(\)](#)) for alignment requirements of packets.

The Programmed I/O region can hold multiple smaller packets, referenced by different offset parameters. All other constraints must still be observed, including:

- alignment
- minimum size
- maximum size
- avoiding reuse until transmission is complete.



### 11.15.2.5 ef\_pio\_unlink\_vi()

```
int ef_pio_unlink_vi (
    ef_pio * pio,
    ef_driver_handle pio_dh,
    struct ef_vi * vi,
    ef_driver_handle vi_dh )
```

Unlink a Programmed I/O region from a virtual interface.

#### Parameters

<i>pio</i>	The Programmed I/O region.
<i>pio_dh</i>	The ef_driver_handle for the Programmed I/O region.
<i>vi</i>	The virtual interface to unlink from the Programmed I/O region.
<i>vi_dh</i>	The ef_driver_handle for the virtual interface.

#### Returns

0 on success, or a negative error code.

Unlink a Programmed I/O region from a virtual interface.

### 11.15.2.6 ef\_vi\_get\_pio\_size()

```
int ef_vi_get_pio_size (
    ef_vi * vi )
```

Get the size of the Programmed I/O region.

#### Parameters

<i>vi</i>	The virtual interface to query.
-----------	---------------------------------

#### Returns

The size of the Programmed I/O region.

Get the size of the Programmed I/O region.

## 11.16 timer.h File Reference

Timers for EtherFabric Virtual Interface HAL.

## Macros

- #define `ef_eventq_timer_prime(q, v)` (q)->ops.eventq\_timer\_prime(q, v)  
*Prime an event queue timer with a new timeout.*
- #define `ef_eventq_timer_run(q, v)` (q)->ops.eventq\_timer\_run(q, v)  
*Start an event queue timer running.*
- #define `ef_eventq_timer_clear(q)` (q)->ops.eventq\_timer\_clear(q)  
*Stop an event queue timer.*
- #define `ef_eventq_timer_zero(q)` (q)->ops.eventq\_timer\_zero(q)  
*Prime an event queue timer to expire immediately.*

### 11.16.1 Detailed Description

Timers for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.16.2 Macro Definition Documentation

#### 11.16.2.1 ef\_eventq\_timer\_clear

```
#define ef_eventq_timer_clear(  
    q ) (q)->ops.eventq_timer_clear(q)
```

Stop an event queue timer.

#### Parameters

<code>q</code>	Pointer to <code>ef_vi</code> structure for the event queue.
----------------	--

#### Returns

None.

Stop an event queue timer.

The timer is stopped if it is already running, and no timeout-event is delivered.

The timer will not run when the next event arrives on the event queue.

#### Note

This is implemented as a macro, that calls the relevant function from the `ef_vi::ops` structure.

Definition at line 113 of file timer.h.

### 11.16.2.2 ef\_eventq\_timer\_prime

```
#define ef_eventq_timer_prime(  
    q,  
    v ) (q)->ops.eventq_timer_prime(q, v)
```

Prime an event queue timer with a new timeout.

#### Parameters

<i>q</i>	Pointer to <a href="#">ef_vi</a> structure for the event queue.
<i>v</i>	Initial value for timer (specified in $\mu$ s).

#### Returns

None.

Prime an event queue timer with a new timeout.

The timer is stopped if it is already running, and no timeout-event is delivered.

The specified timeout is altered slightly, to avoid lots of timers going off in the same tick (bug1317). The timer is then primed with this new timeout.

The timer is then ready to run when the next event arrives on the event queue. When the timer-value reaches zero, a timeout-event will be delivered.

#### Note

This is implemented as a macro, that calls the relevant function from the [ef\\_vi::ops](#) structure.

Definition at line 72 of file timer.h.

### 11.16.2.3 ef\_eventq\_timer\_run

```
#define ef_eventq_timer_run(  
    q,  
    v ) (q)->ops.eventq_timer_run(q, v)
```

Start an event queue timer running.

#### Parameters

<i>q</i>	Pointer to <a href="#">ef_vi</a> structure for the event queue.
<i>v</i>	Initial value for timer (specified in $\mu$ s).

## Returns

None.

Start an event queue timer running.

The timer is stopped if it is already running, and no timeout-event is delivered.

The specified timeout is altered slightly, to avoid lots of timers going off in the same tick (bug1317). The timer is then primed with this new timeout., and starts running immediately.

When the timer-value reaches zero, a timeout-event will be delivered.

## Note

This is implemented as a macro, that calls the relevant function from the [ef\\_vi::ops](#) structure.

Definition at line 95 of file timer.h.

### 11.16.2.4 ef\_eventq\_timer\_zero

```
#define ef_eventq_timer_zero(  
    q ) (q)->ops.eventq_timer_zero(q)
```

Prime an event queue timer to expire immediately.

#### Parameters

<i>q</i>	Pointer to <a href="#">ef_vi</a> structure for the event queue.
----------	---

## Returns

None.

Prime an event queue timer to expire immediately.

The timer is stopped if it is already running, and no timeout-event is delivered.

The timer is then primed with a new timeout of 0.

When the next event arrives on the event queue, a timeout-event will be delivered.

## Note

This is implemented as a macro, that calls the relevant function from the [ef\\_vi::ops](#) structure.

Definition at line 134 of file timer.h.

## 11.17 vi.h File Reference

Virtual packet / DMA interface for EtherFabric Virtual Interface HAL.

```
#include <etherfabric/ef_vi.h>
#include <etherfabric/base.h>
```

### Data Structures

- struct [ef\\_vi\\_set](#)  
*A virtual interface set within a protection domain.*
- struct [ef\\_filter\\_spec](#)  
*Specification of a filter.*
- struct [ef\\_filter\\_cookie](#)  
*Cookie identifying a filter.*
- struct [ef\\_vi\\_stats\\_field\\_layout](#)  
*Layout for a field of statistics.*
- struct [ef\\_vi\\_stats\\_layout](#)  
*Layout for statistics.*

### Enumerations

- enum [ef\\_filter\\_flags](#) { [EF\\_FILTER\\_FLAG\\_NONE](#) = 0x0, [EF\\_FILTER\\_FLAG\\_MCAST\\_LOOP\\_RECEIVE](#) = 0x2 }
  - enum { [EF\\_FILTER\\_VLAN\\_ID\\_ANY](#) = -1 }
- Flags for a filter.*  
*Virtual LANs for a filter.*

### Functions

- int [ef\\_vi\\_alloc\\_from\\_pd](#) ([ef\\_vi](#) \*vi, [ef\\_driver\\_handle](#) vi\_dh, struct [ef\\_pd](#) \*pd, [ef\\_driver\\_handle](#) pd\_dh, int evq\_capacity, int rxq\_capacity, int txq\_capacity, [ef\\_vi](#) \*evq\_opt, [ef\\_driver\\_handle](#) evq\_dh, enum [ef\\_vi\\_flags](#) flags)  
*Allocate a virtual interface from a protection domain.*
- int [ef\\_vi\\_free](#) ([ef\\_vi](#) \*vi, [ef\\_driver\\_handle](#) nic)  
*Free a virtual interface.*
- int [ef\\_vi\\_transmit\\_alt\\_alloc](#) (struct [ef\\_vi](#) \*vi, [ef\\_driver\\_handle](#) vi\_dh, int num\_alts, size\_t buf\_space)  
*Allocate a set of TX alternatives.*
- int [ef\\_vi\\_transmit\\_alt\\_free](#) (struct [ef\\_vi](#) \*vi, [ef\\_driver\\_handle](#) vi\_dh)  
*Free a set of TX alternatives.*
- int [ef\\_vi\\_transmit\\_alt\\_query\\_buffering](#) (struct [ef\\_vi](#) \*vi, int ifindex, [ef\\_driver\\_handle](#) vi\_dh, int n\_alts)  
*Query available buffering.*
- int [ef\\_vi\\_flush](#) ([ef\\_vi](#) \*vi, [ef\\_driver\\_handle](#) nic)  
*Flush the virtual interface.*
- int [ef\\_vi\\_pace](#) ([ef\\_vi](#) \*vi, [ef\\_driver\\_handle](#) nic, int val)  
*Pace the virtual interface.*

- unsigned `ef_vi_mtu` (`ef_vi *vi`, `ef_driver_handle vi_dh`)  
*Return the virtual interface MTU.*
- int `ef_vi_get_mac` (`ef_vi *vi`, `ef_driver_handle vi_dh`, void `*mac_out`)  
*Get the Ethernet MAC address for the virtual interface.*
- int `ef_eventq_put` (unsigned `resource_id`, `ef_driver_handle evq_dh`, unsigned `ev_bits`)  
*Send a software-generated event to an event queue.*
- int `ef_vi_set_alloc_from_pd` (`ef_vi_set *vi_set`, `ef_driver_handle vi_set_dh`, struct `ef_pd *pd`, `ef_driver_handle pd_dh`, int `n_vis`)  
*Allocate a virtual interface set within a protection domain.*
- int `ef_vi_set_free` (`ef_vi_set *vi_set`, `ef_driver_handle vi_set_dh`)  
*Free a virtual interface set.*
- int `ef_vi_alloc_from_set` (`ef_vi *vi`, `ef_driver_handle vi_dh`, `ef_vi_set *vi_set`, `ef_driver_handle vi_set_dh`, int `index_in_vi_set`, int `evq_capacity`, int `rxq_capacity`, int `txq_capacity`, `ef_vi *evq_opt`, `ef_driver_handle evq_dh`, enum `ef_vi_flags flags`)  
*Allocate a virtual interface from a virtual interface set.*
- int `ef_vi_prime` (`ef_vi *vi`, `ef_driver_handle dh`, unsigned `current_ptr`)  
*Prime a virtual interface.*
- void `ef_filter_spec_init` (`ef_filter_spec *filter_spec`, enum `ef_filter_flags flags`)  
*Initialize an ef\_filter\_spec.*
- int `ef_filter_spec_set_ip4_local` (`ef_filter_spec *filter_spec`, int `protocol`, unsigned `host_be32`, int `port_be16`)  
*Set an IP4 Local filter on the filter specification.*
- int `ef_filter_spec_set_ip4_full` (`ef_filter_spec *filter_spec`, int `protocol`, unsigned `host_be32`, int `port_be16`, unsigned `rhost_be32`, int `rport_be16`)  
*Set an IP4 Full filter on the filter specification.*
- int `ef_filter_spec_set_ip6_local` (`ef_filter_spec *filter_spec`, int `protocol`, const struct `in6_addr *host`, int `port_be16`)  
*Set an IP6 Local filter on the filter specification.*
- int `ef_filter_spec_set_ip6_full` (`ef_filter_spec *filter_spec`, int `protocol`, const struct `in6_addr *host`, int `port_be16`, const struct `in6_addr *rhost`, int `rport_be16`)  
*Set an IP6 Full filter on the filter specification.*
- int `ef_filter_spec_set_vlan` (`ef_filter_spec *filter_spec`, int `vlan_id`)  
*Add a Virtual LAN filter on the filter specification.*
- int `ef_filter_spec_set_eth_local` (`ef_filter_spec *filter_spec`, int `vlan_id`, const void `*mac`)  
*Set an Ethernet MAC Address filter on the filter specification.*
- int `ef_filter_spec_set_unicast_all` (`ef_filter_spec *filter_spec`)  
*Set a Unicast All filter on the filter specification.*
- int `ef_filter_spec_set_multicast_all` (`ef_filter_spec *filter_spec`)  
*Set a Multicast All filter on the filter specification.*
- int `ef_filter_spec_set_unicast_mismatch` (`ef_filter_spec *filter_spec`)  
*Set a Unicast Mismatch filter on the filter specification.*
- int `ef_filter_spec_set_multicast_mismatch` (`ef_filter_spec *filter_spec`)  
*Set a Multicast Mismatch filter on the filter specification.*
- int `ef_filter_spec_set_port_sniff` (`ef_filter_spec *filter_spec`, int `promiscuous`)  
*Set a Port Sniff filter on the filter specification.*
- int `ef_filter_spec_set_tx_port_sniff` (`ef_filter_spec *filter_spec`)  
*Set a TX Port Sniff filter on the filter specification.*
- int `ef_filter_spec_set_block_kernel` (`ef_filter_spec *filter_spec`)  
*Set a Block Kernel filter on the filter specification.*
- int `ef_filter_spec_set_block_kernel_multicast` (`ef_filter_spec *filter_spec`)

- Set a Block Kernel Multicast filter on the filter specification.*

  - int `ef_filter_spec_set_eth_type` (`ef_filter_spec *filter_spec`, `uint16_t ether_type_be16`)

*Add an EtherType filter on the filter specification.*
- int `ef_filter_spec_set_ip_proto` (`ef_filter_spec *filter_spec`, `uint8_t ip_proto`)
- Add an IP protocol filter on the filter specification.*

  - int `ef_filter_spec_set_block_kernel_unicast` (`ef_filter_spec *filter_spec`)

*Set a Block Kernel Unicast filter on the filter specification.*
- int `ef_vi_filter_add` (`ef_vi *vi`, `ef_driver_handle vi_dh`, `const ef_filter_spec *filter_spec`, `ef_filter_cookie *filter_cookie_out`)
- Add a filter to a virtual interface.*

  - int `ef_vi_filter_del` (`ef_vi *vi`, `ef_driver_handle vi_dh`, `ef_filter_cookie *filter_cookie`)

*Delete a filter from a virtual interface.*
- int `ef_vi_set_filter_add` (`ef_vi_set *vi_set`, `ef_driver_handle vi_set_dh`, `const ef_filter_spec *filter_spec`, `ef_filter_cookie *filter_cookie_out`)
- Add a filter to a virtual interface set.*

  - int `ef_vi_set_filter_del` (`ef_vi_set *vi_set`, `ef_driver_handle vi_set_dh`, `ef_filter_cookie *filter_cookie`)

*Delete a filter from a virtual interface set.*
- int `ef_vi_stats_query_layout` (`ef_vi *vi`, `const ef_vi_stats_layout **const layout_out`)
- Retrieve layout for available statistics.*

  - int `ef_vi_stats_query` (`ef_vi *vi`, `ef_driver_handle vi_dh`, `void *data`, `int do_reset`)

*Retrieve a set of statistic values.*

### 11.17.1 Detailed Description

Virtual packet / DMA interface for EtherFabric Virtual Interface HAL.

#### Author

Solarflare Communications, Inc.

#### Date

2018/11/06

#### Copyright

Copyright © 2018 Solarflare Communications, Inc. All rights reserved. Solarflare, OpenOnload and EnterpriseOnload are trademarks of Solarflare Communications, Inc.

### 11.17.2 Enumeration Type Documentation

#### 11.17.2.1 anonymous enum

anonymous enum

Virtual LANs for a filter.

### Enumerator

EF_FILTER_VLAN_ID_ANY	Any Virtual LAN
-----------------------	-----------------

Definition at line 479 of file vi.h.

### 11.17.2.2 ef\_filter\_flags

enum `ef_filter_flags`

Flags for a filter.

### Enumerator

EF_FILTER_FLAG_NONE	No flags
EF_FILTER_FLAG_MCAST_LOOP_RECEIVE	If set, the filter will receive looped back packets for matching (see <a href="#">ef_filter_spec_set_tx_port_sniff()</a> )

Definition at line 460 of file vi.h.

## 11.17.3 Function Documentation

### 11.17.3.1 ef\_eventq\_put()

```
int ef_eventq_put (
    unsigned resource_id,
    ef_driver_handle evq_dh,
    unsigned ev_bits )
```

Send a software-generated event to an event queue.

### Parameters

<i>resource_id</i>	The ID of the event queue.
<i>evq_dh</i>	The <code>ef_driver_handle</code> for the event queue.
<i>ev_bits</i>	Data for the event. The lowest 16 bits only are used, and all other bits must be clear.

### Returns

0 on success, or a negative error code.

Send a software-generated event to an event queue.

An application can use this feature to put its own signals onto the event queue. For example, a thread might block waiting for events. An application could use a software-generated event to wake up the thread, so the thread could then process some non-ef\_vi resources.



### 11.17.3.2 ef\_filter\_spec\_init()

```
void ef_filter_spec_init (
    ef_filter_spec * filter_spec,
    enum ef_filter_flags flags )
```

Initialize an `ef_filter_spec`.

### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> to initialize.
<i>flags</i>	The flags to set in the <a href="#">ef_filter_spec</a> .

### Returns

None.

Initialize an [ef\\_filter\\_spec](#).

This function must be called to initialize a filter before calling the other filter functions.

The EF\_FILTER\_FLAG\_MCAST\_LOOP\_RECEIVE flag does the following:

- if set, the filter will receive looped back packets for matching (see [ef\\_filter\\_spec\\_set\\_tx\\_port\\_sniff\(\)](#))
- otherwise, the filter will not receive looped back packets.

#### 11.17.3.3 ef\_filter\_spec\_set\_block\_kernel()

```
int ef_filter_spec_set_block_kernel (
    ef_filter_spec * filter_spec )
```

Set a Block Kernel filter on the filter specification.

### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Block Kernel filter on the filter specification.

This filter blocks all packets from reaching the kernel.

This filter is not supported by 5000-series and 6000-series adapters.

#### 11.17.3.4 ef\_filter\_spec\_set\_block\_kernel\_multicast()

```
int ef_filter_spec_set_block_kernel_multicast (
    ef_filter_spec * filter_spec )
```

Set a Block Kernel Multicast filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Block Kernel Multicast filter on the filter specification.

This filter blocks all multicast packets from reaching the kernel.

This filter is not supported by 5000-series and 6000-series adapters.

#### 11.17.3.5 ef\_filter\_spec\_set\_block\_kernel\_unicast()

```
int ef_filter_spec_set_block_kernel_unicast (
    ef_filter_spec * filter_spec )
```

Set a Block Kernel Unicast filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Block Kernel Unicast filter on the filter specification.

This filter blocks all unicast packets from reaching the kernel.

This filter is not supported by 5000-series and 6000-series adapters.

#### 11.17.3.6 ef\_filter\_spec\_set\_eth\_local()

```
int ef_filter_spec_set_eth_local (
    ef_filter_spec * filter_spec,
    int vlan_id,
    const void * mac )
```

Set an Ethernet MAC Address filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>vlan_id</i>	The ID of the virtual LAN on which to filter, or EF_FILTER_VLAN_ID_ANY to match all VLANs.
<i>mac</i>	The MAC address on which to filter, as a six-byte array.

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set an Ethernet MAC Address filter on the filter specification.

This filter intercepts all packets that match the given MAC address and VLAN.

### 11.17.3.7 ef\_filter\_spec\_set\_eth\_type()

```
int ef_filter_spec_set_eth_type (
    ef_filter_spec * filter_spec,
    uint16_t ether_type_be16 )
```

Add an EtherType filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>ether_type_be16</i>	The EtherType on which to filter, in network order.

#### Returns

- 0 on success, or a negative error code:
- EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Add an EtherType filter on the filter specification

The EtherType filter can be combined with other filters as follows:

- Ethernet MAC Address filters: supported. See [ef\\_filter\\_spec\\_set\\_eth\\_local\(\)](#).
- Other filters: not supported, -EPROTONOSUPPORT is returned.

This filter is not supported by 5000-series and 6000-series adapters. 7000-series adapters require a firmware version of at least v4.6 for full support for these filters. v4.5 firmware supports such filters only when not combined with a MAC address. Insertion of such filters on firmware versions that do not support them will fail.

Due to a current firmware limitation, this method does not support ether\_type IP or IPv6 and will return no error if these values are specified.

### 11.17.3.8 ef\_filter\_spec\_set\_ip4\_full()

```
int ef_filter_spec_set_ip4_full (
    ef_filter_spec * filter_spec,
    int protocol,
    unsigned host_be32,
    int port_be16,
    unsigned rhost_be32,
    int rport_be16 )
```

Set an IP4 Full filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>protocol</i>	The protocol on which to filter (IPPROTO_UDP or IPPROTO_TCP).
<i>host_be32</i>	The local host address on which to filter, as a 32-bit big-endian value (e.g. the output of htonl()).
<i>port_be16</i>	The local port on which to filter, as a 16-bit big-endian value (e.g. the output of htons()).
<i>rhost_be32</i>	The remote host address on which to filter, as a 32-bit big-endian value (e.g. the output of htonl()).
<i>rport_be16</i>	The remote port on which to filter, as a 16-bit big-endian value (e.g. the output of htons()).

## Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set an IP4 Full filter on the filter specification.

This filter intercepts all packets that match the given protocol and host/port combinations.

## Note

You cannot specify a range, or a wildcard, for any parameter.

### 11.17.3.9 ef\_filter\_spec\_set\_ip4\_local()

```
int ef_filter_spec_set_ip4_local (
    ef_filter_spec * filter_spec,
    int protocol,
    unsigned host_be32,
    int port_be16 )
```

Set an IP4 Local filter on the filter specification.

Set various types of filters on the filter spec

## Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>protocol</i>	The protocol on which to filter (IPPROTO_UDP or IPPROTO_TCP).
<i>host_be32</i>	The local host address on which to filter, as a 32-bit big-endian value (e.g. the output of htonl()).
<i>port_be16</i>	The local port on which to filter, as a 16-bit big-endian value (e.g. the output of htons()).

## Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set an IP4 Local filter on the filter specification.

This filter intercepts all packets that match the given protocol and host/port combination.

## Note

You cannot specify a range, or a wildcard, for any parameter.

### 11.17.3.10 ef\_filter\_spec\_set\_ip6\_full()

```
int ef_filter_spec_set_ip6_full (
    ef_filter_spec * filter_spec,
    int protocol,
    const struct in6_addr * host,
    int port_be16,
    const struct in6_addr * rhost,
    int rport_be16 )
```

Set an IP6 Full filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>protocol</i>	The protocol on which to filter (IPPROTO_UDP or IPPROTO_TCP).
<i>host</i>	The local host address on which to filter, as a pointer to a struct <code>in6_addr</code> .
<i>port_be16</i>	The local port on which to filter, as a 16-bit big-endian value (e.g. the output of <code>htons()</code> ).
<i>rhost</i>	The remote host address on which to filter, as a pointer to a struct <code>in6_addr</code> .
<i>rport_be16</i>	The remote port on which to filter, as a 16-bit big-endian value (e.g. the output of <code>htons()</code> ).

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set an IP6 Full filter on the filter specification.

This filter intercepts all packets that match the given protocol and host/port combinations.

#### Note

You cannot specify a range, or a wildcard, for any parameter.

#### 11.17.3.11 ef\_filter\_spec\_set\_ip6\_local()

```
int ef_filter_spec_set_ip6_local (
    ef_filter_spec * filter_spec,
    int protocol,
    const struct in6_addr * host,
    int port_be16 )
```

Set an IP6 Local filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>protocol</i>	The protocol on which to filter (IPPROTO_UDP or IPPROTO_TCP).
<i>host</i>	The local host address on which to filter, as a pointer to a struct <code>in6_addr</code> .
<i>port_be16</i>	The local port on which to filter, as a 16-bit big-endian value (e.g. the output of <code>htons()</code> ).

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set an IP6 Local filter on the filter specification.

This filter intercepts all packets that match the given protocol and host/port combination.

#### Note

You cannot specify a range, or a wildcard, for any parameter.

### 11.17.3.12 ef\_filter\_spec\_set\_ip\_proto()

```
int ef_filter_spec_set_ip_proto (
    ef_filter_spec * filter_spec,
    uint8_t ip_proto )
```

Add an IP protocol filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>ip_proto</i>	The IP protocol on which to filter.

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Add an IP protocol filter on the filter specification

The IP protocol filter can be combined with other filters as follows:

- Ethernet MAC Address filters: supported. See [ef\\_filter\\_spec\\_set\\_eth\\_local\(\)](#).
- Other filters: not supported, -EPROTONOSUPPORT is returned.

This filter is not supported by 5000-series and 6000-series adapters. Other adapters require a firmware version of at least v4.5. Insertion of such filters on firmware versions that do not support them will fail.

Due to a current firmware limitation, this method does not support `ip_proto=6` (TCP) or `ip_proto=17` (UDP) and will return no error if these values are used.

### 11.17.3.13 ef\_filter\_spec\_set\_multicast\_all()

```
int ef_filter_spec_set_multicast_all (
    ef_filter_spec * filter_spec )
```

Set a Multicast All filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Multicast All filter on the filter specification.

This filter must be used with caution. It intercepts all multicast packets that arrive, including IGMP group membership queries, which must normally be handled by the kernel to avoid any membership lapses.

### 11.17.3.14 ef\_filter\_spec\_set\_multicast\_mismatch()

```
int ef_filter_spec_set_multicast_mismatch (
    ef_filter_spec * filter_spec )
```

Set a Multicast Mismatch filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Multicast Mismatch filter on the filter specification.

This filter intercepts all multicast traffic that would otherwise be discarded; that is, all traffic that does not match either an existing multicast filter or a kernel subscription.

This filter is not supported by 5000-series and 6000-series adapters.

#### 11.17.3.15 ef\_filter\_spec\_set\_port\_sniff()

```
int ef_filter_spec_set_port_sniff (
    ef_filter_spec * filter_spec,
    int promiscuous )
```

Set a Port Sniff filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>promiscuous</i>	True to enable promiscuous mode on any virtual interface using this filter.

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Port Sniff filter on the filter specification.

This filter enables sniff mode for the virtual interface. All filtering on that interface then copies packets instead of intercepting them. Consequently, the kernel receives the filtered packets; otherwise it would not.

If promiscuous mode is enabled, this filter copies all packets, instead of only those matched by other filters.

This filter is not supported by 5000-series and 6000-series adapters.

#### 11.17.3.16 ef\_filter\_spec\_set\_tx\_port\_sniff()

```
int ef_filter_spec_set_tx_port_sniff (
    ef_filter_spec * filter_spec )
```

Set a TX Port Sniff filter on the filter specification.



#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a TX Port Sniff filter on the filter specification.

This filter loops back a copy of all outgoing packets, so that your application can process them.

This filter is not supported by 5000-series and 6000-series adapters.

#### 11.17.3.17 ef\_filter\_spec\_set\_unicast\_all()

```
int ef_filter_spec_set_unicast_all (
    ef_filter_spec * filter_spec )
```

Set a Unicast All filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Unicast All filter on the filter specification.

This filter must be used with caution. It intercepts all unicast packets that arrive, including ARP resolutions, which must normally be handled by the kernel for routing to work.

#### 11.17.3.18 ef\_filter\_spec\_set\_unicast\_mismatch()

```
int ef_filter_spec_set_unicast_mismatch (
    ef_filter_spec * filter_spec )
```

Set a Unicast Mismatch filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
--------------------	--

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Set a Unicast Mismatch filter on the filter specification.

This filter intercepts all unicast traffic that would otherwise be discarded; that is, all traffic that does not match either an existing unicast filter or a kernel subscription.

This filter is not supported by 5000-series and 6000-series adapters.

### 11.17.3.19 ef\_filter\_spec\_set\_vlan()

```
int ef_filter_spec_set_vlan (
    ef_filter_spec * filter_spec,
    int vlan_id )
```

Add a Virtual LAN filter on the filter specification.

#### Parameters

<i>filter_spec</i>	The <a href="#">ef_filter_spec</a> on which to set the filter.
<i>vlan_id</i>	The ID of the virtual LAN on which to filter.

#### Returns

0 on success, or a negative error code:  
-EPROTONOSUPPORT indicates that a filter is already set that is incompatible with the new filter.

Add a Virtual LAN filter on the filter specification.

The Virtual LAN filter can be combined with other filters as follows:

- Ethernet MAC Address filters: supported. See [ef\\_filter\\_spec\\_set\\_eth\\_local\(\)](#).
- EtherType filters: supported.
- IP protocol filters: supported.
- IP4 filters:
  - 7000-series adapter with full feature firmware: supported. Packets that match the IP4 filter will be received only if they also match the VLAN.
  - Otherwise: not supported. Packets that match the IP4 filter will always be received, whatever the VLAN.
- Other filters: not supported, -EPROTONOSUPPORT is returned.

### 11.17.3.20 ef\_vi\_alloc\_from\_pd()

```
int ef_vi_alloc_from_pd (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    struct ef_pd * pd,
    ef_driver_handle pd_dh,
    int evq_capacity,
    int rxq_capacity,
    int txq_capacity,
    ef_vi * evq_opt,
    ef_driver_handle evq_dh,
    enum ef_vi_flags flags )
```

Allocate a virtual interface from a protection domain.

### Parameters

<i>vi</i>	Memory for the allocated virtual interface.
<i>vi_dh</i>	The ef_driver_handle to associate with the virtual interface.
<i>pd</i>	The protection domain from which to allocate the virtual interface.
<i>pd_dh</i>	The ef_driver_handle to associate with the protection domain.
<i>evq_capacity</i>	The capacity of the event queue, or: <ul style="list-style-type: none"> <li>• 0 for no event queue</li> <li>• -1 for the default size (EF_VI_EVQ_SIZE if set, otherwise it is rxq_capacity + txq_capacity + extra bytes if timestamps are enabled).</li> </ul>
<i>rxq_capacity</i>	The number of slots in the RX descriptor ring, or: <ul style="list-style-type: none"> <li>• 0 for no event queue</li> <li>• -1 for the default size (EF_VI_RXQ_SIZE if set, otherwise 512).</li> </ul>
<i>txq_capacity</i>	The number of slots in the TX descriptor ring, or: <ul style="list-style-type: none"> <li>• 0 for no TX descriptor ring</li> <li>• -1 for the default size (EF_VI_TXQ_SIZE if set, otherwise 512).</li> </ul>
<i>evq_opt</i>	event queue to use if evq_capacity=0.
<i>evq_dh</i>	The ef_driver_handle of the evq_opt event queue.
<i>flags</i>	Flags to select hardware attributes of the virtual interface.

### Returns

>= 0 on success (value is Q\_ID), or a negative error code.

Allocate a virtual interface from a protection domain.

This allocates an RX and TX descriptor ring, an event queue, timers and interrupt etc. on the card. It also initializes the (opaque) structures needed to access them in software.

An existing virtual interface can be specified, to resize its descriptor rings and event queue.

When setting the sizes of the descriptor rings and event queue for a new or existing virtual interface:

- the event queue should be left at its default size unless extra rings are added
- if extra descriptor rings are added, the event queue should also be made correspondingly larger
- the maximum size of the event queue effectively limits how many descriptor ring slots can be supported without risking the event queue overflowing.

### 11.17.3.21 ef\_vi\_alloc\_from\_set()

```
int ef_vi_alloc_from_set (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    ef_vi_set * vi_set,
    ef_driver_handle vi_set_dh,
    int index_in_vi_set,
    int evq_capacity,
    int rxq_capacity,
    int txq_capacity,
    ef_vi * evq_opt,
    ef_driver_handle evq_dh,
    enum ef_vi_flags flags )
```

Allocate a virtual interface from a virtual interface set.

#### Parameters

<i>vi</i>	Memory for the allocated virtual interface.
<i>vi_dh</i>	The ef_driver_handle to associate with the virtual interface.
<i>vi_set</i>	The virtual interface set from which to allocate the virtual interface.
<i>vi_set_dh</i>	The ef_driver_handle to associate with the virtual interface set.
<i>index_in_vi_set</i>	Index of the virtual interface within the set to allocate, or -1 for any.
<i>evq_capacity</i>	The number of events in the event queue (maximum 32768), or: <ul style="list-style-type: none"> <li>• 0 for no event queue</li> <li>• -1 for the default size.</li> </ul>
<i>rxq_capacity</i>	The number of slots in the RX descriptor ring, or: <ul style="list-style-type: none"> <li>• 0 for no RX queue</li> <li>• -1 for the default size (EF_VI_RXQ_SIZE if set, otherwise 512).</li> </ul>
<i>txq_capacity</i>	The number of slots in the TX descriptor ring, or: <ul style="list-style-type: none"> <li>• 0 for no TX queue</li> <li>• -1 for the default size (EF_VI_TXQ_SIZE if set, otherwise 512).</li> </ul>
<i>evq_opt</i>	event queue to use if evq_capacity=0.
<i>evq_dh</i>	The ef_driver_handle of the evq_opt event queue.
<i>flags</i>	Flags to select hardware attributes of the virtual interface.

#### Returns

>= 0 on success (value is Q\_ID), or a negative error code.

Allocate a virtual interface from a virtual interface set.

This allocates an RX and TX descriptor ring, an event queue, timers and interrupt etc. on the card. It also initializes the (opaque) structures needed to access them in software.

An existing virtual interface can be specified, to resize its descriptor rings and event queue.

When setting the sizes of the descriptor rings and event queue for a new or existing virtual interface:

- the event queue should be left at its default size unless extra rings are added
- if extra descriptor rings are added, the event queue should also be made correspondingly larger
- the maximum size of the event queue effectively limits how many descriptor ring slots can be supported without risking the event queue overflowing.

### 11.17.3.22 ef\_vi\_filter\_add()

```
int ef_vi_filter_add (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    const ef_filter_spec * filter_spec,
    ef_filter_cookie * filter_cookie_out )
```

Add a filter to a virtual interface.

#### Parameters

<i>vi</i>	The virtual interface on which to add the filter.
<i>vi_dh</i>	The <code>ef_driver_handle</code> for the virtual interface.
<i>filter_spec</i>	The filter to add.
<i>filter_cookie_out</i>	Optional pointer to an <code>ef_filter_cookie</code> , that is updated on return with a cookie for the filter.

#### Returns

0 on success, or a negative error code.

Add a filter to a virtual interface.

`filter_cookie_out` can be NULL. If not null, then the returned value can be used in `ef_vi_filter_del()` to remove this filter.

After calling this function, any local copy of the filter can be deleted.

### 11.17.3.23 ef\_vi\_filter\_del()

```
int ef_vi_filter_del (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    ef_filter_cookie * filter_cookie )
```

Delete a filter from a virtual interface.

#### Parameters

<i>vi</i>	The virtual interface from which to delete the filter.
<i>vi_dh</i>	The ef_driver_handle for the virtual interface.
<i>filter_cookie</i>	The filter cookie for the filter to delete, as set on return from <a href="#">ef_vi_filter_add()</a> .

#### Returns

0 on success, or a negative error code.

Delete a filter from a virtual interface.

#### 11.17.3.24 ef\_vi\_flush()

```
int ef_vi_flush (
    ef_vi * vi,
    ef_driver_handle nic )
```

Flush the virtual interface.

#### Parameters

<i>vi</i>	The virtual interface to flush.
<i>nic</i>	The ef_driver_handle for the NIC hosting the interface.

#### Returns

0 on success, or a negative error code.

Flush the virtual interface.

After this function returns, it is safe to reuse all buffers which have been pushed onto the NIC.

#### 11.17.3.25 ef\_vi\_free()

```
int ef_vi_free (
    ef_vi * vi,
    ef_driver_handle nic )
```

Free a virtual interface.

#### Parameters

<i>vi</i>	The virtual interface to free.
<i>nic</i>	The ef_driver_handle for the NIC hosting the interface.

## Returns

0 on success, or a negative error code.

Free a virtual interface.

This should be called when a virtual interface is no longer needed.

To free up all resources, you must also close the associated driver handle using `ef_driver_close` and free up memory from the protection domain `ef_pd_free`. See [Freeing Resources](#)

If successful:

- the memory for state provided for this virtual interface is no longer required
- no further events from this virtual interface will be delivered to its event queue.

### 11.17.3.26 ef\_vi\_get\_mac()

```
int ef_vi_get_mac (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    void * mac_out )
```

Get the Ethernet MAC address for the virtual interface.

#### Parameters

<i>vi</i>	The virtual interface to query.
<i>vi_dh</i>	The <code>ef_driver_handle</code> for the NIC hosting the interface.
<i>mac_out</i>	Pointer to a six-byte buffer, that is updated on return with the Ethernet MAC address.

## Returns

0 on success, or a negative error code.

Get the Ethernet MAC address for the virtual interface.

This is not a cheap call, so cache the result if you care about performance.

### 11.17.3.27 ef\_vi\_mtu()

```
unsigned ef_vi_mtu (
    ef_vi * vi,
    ef_driver_handle vi_dh )
```

Return the virtual interface MTU.

### Parameters

<i>vi</i>	The virtual interface to query.
<i>vi_dh</i>	The <code>ef_driver_handle</code> for the NIC hosting the interface.

### Returns

The virtual interface Maximum Transmission Unit.

Return the virtual interface MTU. (This is the maximum size of Ethernet frames that can be transmitted through, and received by the interface).

The returned value is the total frame size, including all headers, but not including the Ethernet frame check.

### 11.17.3.28 ef\_vi\_pace()

```
int ef_vi_pace (
    ef_vi * vi,
    ef_driver_handle nic,
    int val )
```

Pace the virtual interface.

### Parameters

<i>vi</i>	The virtual interface to pace.
<i>nic</i>	The <code>ef_driver_handle</code> for the NIC hosting the interface.
<i>val</i>	The minimum inter-packet gap for the TXQ.

### Returns

0 on success, or a negative error code.

Pace the virtual interface.

This sets a minimum inter-packet gap for the TXQ:

- if *val* is -1 then the TXQ is put into the "pacing" bin, but no gap is enforced
- otherwise, the gap is  $(2^{\text{val}}) * 100\text{ns}$ .

This can be used to give priority to latency sensitive traffic over bulk traffic.

### 11.17.3.29 ef\_vi\_prime()

```
int ef_vi_prime (
    ef_vi * vi,
    ef_driver_handle dh,
    unsigned current_ptr )
```

Prime a virtual interface.



#### Parameters

<i>vi</i>	The virtual interface to prime.
<i>dh</i>	The <code>ef_driver_handle</code> to associate with the virtual interface.
<i>current_ptr</i>	Value returned from <code>ef_eventq_current()</code> .

#### Returns

0 on success, or a negative error code.

Prime a virtual interface. This enables interrupts so you can block on the file descriptor associated with the `ef_driver_handle` using `select/poll/epoll`, etc.

Passing the current event queue pointer ensures correct handling of any events that occur between this prime and the `epoll_wait` call.

#### 11.17.3.30 ef\_vi\_set\_alloc\_from\_pd()

```
int ef_vi_set_alloc_from_pd (
    ef_vi_set * vi_set,
    ef_driver_handle vi_set_dh,
    struct ef_pd * pd,
    ef_driver_handle pd_dh,
    int n_vis )
```

Allocate a virtual interface set within a protection domain.

#### Parameters

<i>vi_set</i>	Memory for the allocated virtual interface set.
<i>vi_set_dh</i>	The <code>ef_driver_handle</code> to associate with the virtual interface set.
<i>pd</i>	The protection domain from which to allocate the virtual interface set.
<i>pd_dh</i>	The <code>ef_driver_handle</code> of the associated protection domain.
<i>n_vis</i>	The number of virtual interfaces in the virtual interface set.

#### Returns

0 on success, or a negative error code.

Allocate a virtual interface set within a protection domain.

A virtual interface set is usually used to spread the load of handling received packets. This is sometimes called receive-side scaling, or RSS.

#### 11.17.3.31 ef\_vi\_set\_filter\_add()

```
int ef_vi_set_filter_add (
    ef_vi_set * vi_set,
    ef_driver_handle vi_set_dh,
    const ef_filter_spec * filter_spec,
    ef_filter_cookie * filter_cookie_out )
```

Add a filter to a virtual interface set.

### Parameters

<i>vi_set</i>	The virtual interface set on which to add the filter.
<i>vi_set_dh</i>	The ef_driver_handle for the virtual interface set.
<i>filter_spec</i>	The filter to add.
<i>filter_cookie_out</i>	Optional pointer to an ef_filter_cookie, that is updated on return with a cookie for the filter.

### Returns

0 on success, or a negative error code:  
Add a filter to a virtual interface set.

filter\_cookie\_out can be NULL. If not null, then the returned value can be used in ef\_vi\_filter\_del() to delete this filter.

After calling this function, any local copy of the filter can be deleted.

#### 11.17.3.32 ef\_vi\_set\_filter\_del()

```
int ef_vi_set_filter_del (
    ef_vi_set * vi_set,
    ef_driver_handle vi_set_dh,
    ef_filter_cookie * filter_cookie )
```

Delete a filter from a virtual interface set.

### Parameters

<i>vi_set</i>	The virtual interface set from which to delete the filter.
<i>vi_set_dh</i>	The ef_driver_handle for the virtual interface set.
<i>filter_cookie</i>	The filter cookie for the filter to delete.

### Returns

0 on success, or a negative error code.

Delete a filter from a virtual interface set.

#### 11.17.3.33 ef\_vi\_set\_free()

```
int ef_vi_set_free (
    ef_vi_set * vi_set,
    ef_driver_handle vi_set_dh )
```

Free a virtual interface set.

### Parameters

<i>vi_set</i>	Memory for the allocated virtual interface set.
<i>vi_set_dh</i>	The ef_driver_handle to associate with the virtual interface set.

### Returns

0 on success, or a negative error code.

Free a virtual interface set.

To free up all resources, you must also close the associated driver handle.

### 11.17.3.34 ef\_vi\_stats\_query()

```
int ef_vi_stats_query (
    ef_vi * vi,
    ef_driver_handle vi_dh,
    void * data,
    int do_reset )
```

Retrieve a set of statistic values.

#### Parameters

<i>vi</i>	The virtual interface to query.
<i>vi_dh</i>	The ef_driver_handle for the virtual interface.
<i>data</i>	Pointer to a buffer, into which the statistics are retrieved. The size of this buffer should be equal to the evsl_data_bytes field of the layout description, that can be fetched using <a href="#">ef_vi_stats_query_layout()</a> .
<i>do_reset</i>	True to reset the statistics after retrieving them.

#### Returns

0 on success, or a negative error code.

Retrieve a set of statistic values.

If do\_reset is true, the statistics are reset after reading.

#### Note

This requires full feature firmware. If used with low-latency firmware, no error is given, and the statistics are invalid (typically all zeroes).

### 11.17.3.35 ef\_vi\_stats\_query\_layout()

```
int ef_vi_stats_query_layout (
    ef_vi * vi,
    const ef_vi_stats_layout **const layout_out )
```

Retrieve layout for available statistics.

#### Parameters

<i>vi</i>	The virtual interface to query.
<i>layout_out</i>	Pointer to an ef_vi_stats_layout*, that is updated on return with the layout for available statistics.

#### Returns

0 on success, or a negative error code.

Retrieve layout for available statistics.

### 11.17.3.36 ef\_vi\_transmit\_alt\_alloc()

```
int ef_vi_transmit_alt_alloc (
    struct ef_vi * vi,
    ef_driver_handle vi_dh,
    int num_alts,
    size_t buf_space )
```

Allocate a set of TX alternatives.

#### Parameters

<i>vi</i>	The virtual interface that is to use the TX alternatives.
<i>vi_dh</i>	The ef_driver_handle for the NIC hosting the interface.
<i>num_alts</i>	The number of TX alternatives for which to allocate space.
<i>buf_space</i>	The buffer space required for the set of TX alternatives, in bytes.

#### Returns

- 0 Success.
- EINVAL The num\_alts or buf\_space parameters are invalid, or the VI was allocated without EF\_VI\_TX\_ALT set.
- EALREADY A set of TX alternatives has already been allocated for use with this VI.
- EBUSY Insufficient memory was available (either host memory or packet buffers on the adapter); or too many alternatives requested, or alternatives requested on too many distinct VIs.

Allocate a set of TX alternatives for use with a virtual interface. The virtual interface must have been allocated with the EF\_VI\_TX\_ALT flag.

The space remains allocated until [ef\\_vi\\_transmit\\_alt\\_free\(\)](#) is called or the virtual interface is freed.

TX alternatives provide a mechanism to send with very low latency. They work by pre-loading packets into the adapter in advance, and then calling [ef\\_vi\\_transmit\\_alt\\_go\(\)](#) to transmit the packets.

Packets are pre-loaded into the adapter using normal send calls such as [ef\\_vi\\_transmit\(\)](#). Use [ef\\_vi\\_transmit\\_alt\\_select\(\)](#) to select which "alternative" to load the packet into.

Each alternative has three states: STOP, GO and DISCARD. The [ef\\_vi\\_transmit\\_alt\\_stop\(\)](#), [ef\\_vi\\_transmit\\_alt\\_go\(\)](#) and [ef\\_vi\\_transmit\\_alt\\_discard\(\)](#) calls transition between the states. Typically an alternative is placed in the STOP state, selected, pre-loaded with one or more packets, and then later on the critical path placed in the GO state.

When packets are transmitted via TX alternatives, events of type EF\_EVENT\_TYPE\_TX\_ALT are returned to the application. The application is responsible for ensuring that all of the packets in an alternative have been sent before transitioning from GO or DISCARD to the STOP state.

The *buf\_space* parameter gives the amount of buffering to allocate for this set of TX alternatives, in bytes. Note that if this buffering is exceeded then packets sent to TX alternatives may be truncated or dropped, and no error is reported in this case.

### 11.17.3.37 ef\_vi\_transmit\_alt\_free()

```
int ef_vi_transmit_alt_free (
    struct ef_vi * vi,
    ef_driver_handle vi_dh )
```

Free a set of TX alternatives.

#### Parameters

<i>vi</i>	The virtual interface whose alternatives are to be freed.
<i>vi_dh</i>	The <code>ef_driver_handle</code> for the NIC hosting the interface.

#### Returns

0 on success, or a negative error code.

Release the set of TX alternatives allocated by `ef_vi_transmit_alt_alloc()`.

#### 11.17.3.38 ef\_vi\_transmit\_alt\_query\_buffering()

```
int ef_vi_transmit_alt_query_buffering (
    struct ef_vi * vi,
    int ifindex,
    ef_driver_handle vi_dh,
    int n_alts )
```

Query available buffering.

#### Parameters

<i>vi</i>	Interface to be queried
<i>ifindex</i>	The index of the interface that you wish to query. You can use <code>if_nametoindex()</code> to obtain this. This should be the underlying physical interface, rather than a bond, VLAN, or similar.
<i>vi_dh</i>	The <code>ef_driver_handle</code> for the NIC hosting the interface.
<i>n_alts</i>	Intended number of alternatives

#### Returns

-EINVAL if this VI doesn't support alternatives, else the number of bytes available

Owing to per-packet and other overheads, the amount of data which can be stored in TX alternatives is generally slightly less than the amount of memory available on the hardware.

This function allows the caller to find out how much user-visible buffering will be available if the given number of alternatives are allocated on the given VI.



# Index

- 000\_main.dox, [101](#)
- 005\_whats\_new.dox, [101](#)
- 010\_overview.dox, [102](#)
- 020\_concepts.dox, [102](#)
- 030\_apps.dox, [103](#)
- 040\_using.dox, [103](#)
- 050\_examples.dox, [104](#)
  
- added
  - ef\_vi\_rxq\_state, [82](#)
  - ef\_vi\_txq\_state, [92](#)
- arch
  - ef\_vi\_nic\_type, [79](#)
  
- base.h, [104](#)
  - ef\_driver\_close, [105](#)
  - ef\_driver\_open, [106](#)
  - ef\_eventq\_wait, [106](#)
- bytes\_acc
  - ef\_vi\_rxq\_state, [82](#)
  
- capabilities.h, [107](#)
  - ef\_vi\_capabilities\_get, [110](#)
  - ef\_vi\_capabilities\_max, [110](#)
  - ef\_vi\_capabilities\_name, [110](#)
  - ef\_vi\_capability, [108](#)
- capabilities.h
  - EF\_VI\_CAP\_BUFFER\_MODE, [109](#)
  - EF\_VI\_CAP\_CTPIO, [109](#)
  - EF\_VI\_CAP\_EVQ\_SIZES, [109](#)
  - EF\_VI\_CAP\_HW\_MULTICAST\_LOOPBACK, [108](#)
  - EF\_VI\_CAP\_HW\_MULTICAST\_REPLICATION, [108](#)
  - EF\_VI\_CAP\_HW\_RX\_TIMESTAMPING, [108](#)
  - EF\_VI\_CAP\_HW\_TX\_TIMESTAMPING, [108](#)
  - EF\_VI\_CAP\_MAC\_SPOOFING, [109](#)
  - EF\_VI\_CAP\_MAX, [109](#)
  - EF\_VI\_CAP\_MULTICAST\_FILTER\_CHAINING, [109](#)
  - EF\_VI\_CAP\_NIC\_PACE, [109](#)
  - EF\_VI\_CAP\_PACKED\_STREAM, [108](#)
  - EF\_VI\_CAP\_PACKED\_STREAM\_BUFFER\_SIZES, [109](#)
  - EF\_VI\_CAP\_PHYS\_MODE, [109](#)
  - EF\_VI\_CAP\_PIO, [108](#)
  - EF\_VI\_CAP\_PIO\_BUFFER\_COUNT, [108](#)
  - EF\_VI\_CAP\_PIO\_BUFFER\_SIZE, [108](#)
  - EF\_VI\_CAP\_RX\_FILTER\_ETHERTYPE, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_IP4\_PROTO, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_ETH\_LOCAL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_ETH\_LOCAL\_VLAN, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_IP6\_VLAN, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_IP\_VLAN, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_MCAST\_ALL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_MCAST\_MISMATCH, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_SNIFF, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP6\_FULL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP6\_LOCAL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP\_FULL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP\_LOCAL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UCAST\_ALL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UCAST\_MISMATCH, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP6\_FULL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP6\_LOCAL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP\_FULL, [109](#)
  - EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP\_LOCAL, [109](#)
  - EF\_VI\_CAP\_RX\_FORCE\_EVENT\_MERGING, [109](#)
  - EF\_VI\_CAP\_RX\_FW\_VARIANT, [109](#)
  - EF\_VI\_CAP\_RX\_MERGE, [109](#)
  - EF\_VI\_CAP\_RXQ\_SIZES, [109](#)
  - EF\_VI\_CAP\_TX\_ALTERNATIVES, [109](#)
  - EF\_VI\_CAP\_TX\_ALTERNATIVES\_CP\_BUFFER\_SIZE, [109](#)
  - EF\_VI\_CAP\_TX\_ALTERNATIVES\_CP\_BUFFERS, [109](#)
  - EF\_VI\_CAP\_TX\_ALTERNATIVES\_VFIFOS, [109](#)
  - EF\_VI\_CAP\_TX\_FILTER\_TYPE\_SNIFF, [109](#)
  - EF\_VI\_CAP\_TX\_FW\_VARIANT, [109](#)
  - EF\_VI\_CAP\_TX\_PUSH\_ALWAYS, [109](#)
  - EF\_VI\_CAP\_TXQ\_SIZES, [109](#)
  - EF\_VI\_CAP\_VPORTS, [109](#)
  - EF\_VI\_CAP\_ZERO\_RX\_PREFIX, [109](#)

- checksum.h, [111](#)
  - [ef\\_icmpv6\\_checksum](#), [112](#)
  - [ef\\_ip\\_checksum](#), [112](#)
  - [ef\\_tcp\\_checksum](#), [113](#)
  - [ef\\_tcp\\_checksum\\_ip6](#), [113](#)
  - [ef\\_tcp\\_checksum\\_ipx](#), [113](#)
  - [ef\\_udp\\_checksum](#), [114](#)
  - [ef\\_udp\\_checksum\\_ip6](#), [114](#)
  - [ef\\_udp\\_checksum\\_ipx](#), [115](#)
- data
  - [ef\\_filter\\_spec](#), [58](#)
- descriptors
  - [ef\\_vi\\_rxq](#), [81](#)
  - [ef\\_vi\\_txq](#), [91](#)
- EF\_EVENT\_RX\_MULTI\_CONT
  - [ef\\_vi.h](#), [122](#)
- EF\_EVENT\_RX\_MULTI\_SOP
  - [ef\\_vi.h](#), [123](#)
- EF\_EVENT\_RX\_PS\_NEXT\_BUFFER
  - [ef\\_vi.h](#), [123](#)
- EF\_VI\_ALIGN
  - [ef\\_iovec](#), [60](#)
- EF\_VI\_CTPPIO\_CT\_THRESHOLD\_SNF
  - [ef\\_vi.h](#), [124](#)
- EF\_VI\_PS\_FLAG\_BAD\_IP\_CSUM
  - [packedstream.h](#), [159](#)
- ef\_driver\_close
  - [base.h](#), [105](#)
- ef\_driver\_open
  - [base.h](#), [106](#)
- ef\_event, [51](#)
  - [generic](#), [53](#)
  - [rx](#), [53](#)
  - [rx\\_discard](#), [53](#)
  - [rx\\_multi](#), [53](#)
  - [rx\\_multi\\_discard](#), [53](#)
  - [rx\\_no\\_desc\\_trunc](#), [54](#)
  - [rx\\_packed\\_stream](#), [54](#)
  - [sw](#), [54](#)
  - [tx](#), [54](#)
  - [tx\\_alt](#), [54](#)
  - [tx\\_error](#), [54](#)
  - [tx\\_timestamp](#), [54](#)
- ef\_eventq\_capacity
  - [ef\\_vi.h](#), [139](#)
- ef\_eventq\_current
  - [ef\\_vi.h](#), [139](#)
- ef\_eventq\_has\_event
  - [ef\\_vi.h](#), [140](#)
- ef\_eventq\_has\_many\_events
  - [ef\\_vi.h](#), [140](#)
- ef\_eventq\_poll
  - [ef\\_vi.h](#), [123](#)
- ef\_eventq\_prime
  - [ef\\_vi.h](#), [124](#)
- ef\_eventq\_put
  - [vi.h](#), [178](#)
- ef\_eventq\_state, [55](#)
  - [evq\\_clear\\_stride](#), [55](#)
  - [evq\\_ptr](#), [55](#)
  - [sync\\_flags](#), [56](#)
  - [sync\\_timestamp\\_major](#), [56](#)
  - [sync\\_timestamp\\_minimum](#), [56](#)
  - [sync\\_timestamp\\_minor](#), [56](#)
  - [sync\\_timestamp\\_synchronised](#), [56](#)
- ef\_eventq\_timer\_clear
  - [timer.h](#), [172](#)
- ef\_eventq\_timer\_prime
  - [timer.h](#), [172](#)
- ef\_eventq\_timer\_run
  - [timer.h](#), [173](#)
- ef\_eventq\_timer\_zero
  - [timer.h](#), [174](#)
- ef\_eventq\_wait
  - [base.h](#), [106](#)
- ef\_filter\_cookie, [57](#)
  - [filter\\_id](#), [57](#)
  - [filter\\_type](#), [57](#)
- ef\_filter\_flags
  - [vi.h](#), [178](#)
- ef\_filter\_spec, [58](#)
  - [data](#), [58](#)
  - [flags](#), [58](#)
  - [type](#), [59](#)
- ef\_filter\_spec\_init
  - [vi.h](#), [178](#)
- ef\_filter\_spec\_set\_block\_kernel
  - [vi.h](#), [180](#)
- ef\_filter\_spec\_set\_block\_kernel\_multicast
  - [vi.h](#), [180](#)
- ef\_filter\_spec\_set\_block\_kernel\_unicast
  - [vi.h](#), [181](#)
- ef\_filter\_spec\_set\_eth\_local
  - [vi.h](#), [181](#)
- ef\_filter\_spec\_set\_eth\_type
  - [vi.h](#), [181](#)
- ef\_filter\_spec\_set\_ip4\_full
  - [vi.h](#), [182](#)
- ef\_filter\_spec\_set\_ip4\_local
  - [vi.h](#), [183](#)
- ef\_filter\_spec\_set\_ip6\_full
  - [vi.h](#), [183](#)
- ef\_filter\_spec\_set\_ip6\_local
  - [vi.h](#), [184](#)
- ef\_filter\_spec\_set\_ip\_proto
  - [vi.h](#), [184](#)
- ef\_filter\_spec\_set\_multicast\_all
  - [vi.h](#), [185](#)
- ef\_filter\_spec\_set\_multicast\_mismatch
  - [vi.h](#), [185](#)
- ef\_filter\_spec\_set\_port\_sniff



vi.h, [186](#)  
 ef\_filter\_spec\_set\_tx\_port\_sniff  
   vi.h, [186](#)  
 ef\_filter\_spec\_set\_unicast\_all  
   vi.h, [187](#)  
 ef\_filter\_spec\_set\_unicast\_mismatch  
   vi.h, [187](#)  
 ef\_filter\_spec\_set\_vlan  
   vi.h, [187](#)  
 ef\_icmpv6\_checksum  
   checksum.h, [112](#)  
 ef\_iovec, [59](#)  
   EF\_VI\_ALIGN, [60](#)  
   iov\_len, [60](#)  
 ef\_ip\_checksum  
   checksum.h, [112](#)  
 ef\_memreg, [60](#)  
   mr\_dma\_addr, [61](#)  
   mr\_dma\_addr\_base, [61](#)  
 ef\_memreg\_alloc  
   memreg.h, [156](#)  
 ef\_memreg\_dma\_addr  
   memreg.h, [157](#)  
 ef\_memreg\_free  
   memreg.h, [157](#)  
 ef\_packed\_stream\_packet, [61](#)  
   ps\_cap\_len, [62](#)  
   ps\_flags, [62](#)  
   ps\_next\_offset, [62](#)  
   ps\_orig\_len, [62](#)  
   ps\_pkt\_start\_offset, [62](#)  
   ps\_ts\_nsec, [63](#)  
   ps\_ts\_sec, [63](#)  
 ef\_packed\_stream\_packet\_first  
   packedstream.h, [160](#)  
 ef\_packed\_stream\_packet\_next  
   packedstream.h, [160](#)  
 ef\_packed\_stream\_packet\_payload  
   packedstream.h, [160](#)  
 ef\_packed\_stream\_params, [63](#)  
   psp\_buffer\_align, [64](#)  
   psp\_buffer\_size, [64](#)  
   psp\_max\_usable\_buffers, [64](#)  
   psp\_start\_offset, [64](#)  
 ef\_pd, [65](#)  
   pd\_cluster\_dh, [65](#)  
   pd\_cluster\_name, [65](#)  
   pd\_cluster\_sock, [66](#)  
   pd\_cluster\_viset\_index, [66](#)  
   pd\_cluster\_viset\_resource\_id, [66](#)  
   pd\_flags, [66](#)  
   pd\_intf\_name, [66](#)  
   pd\_resource\_id, [67](#)  
 ef\_pd\_alloc  
   pd.h, [164](#)  
 ef\_pd\_alloc\_by\_name  
   pd.h, [164](#)  
 ef\_pd\_alloc\_with\_vport  
   pd.h, [165](#)  
 ef\_pd\_flags  
   pd.h, [163](#)  
 ef\_pd\_free  
   pd.h, [165](#)  
 ef\_pd\_interface\_name  
   pd.h, [166](#)  
 ef\_pio, [67](#)  
   pio\_buffer, [67](#)  
   pio\_io, [68](#)  
   pio\_len, [68](#)  
   pio\_resource\_id, [68](#)  
 ef\_pio\_alloc  
   pio.h, [167](#)  
 ef\_pio\_free  
   pio.h, [169](#)  
 ef\_pio\_link\_vi  
   pio.h, [169](#)  
 ef\_pio\_memcpy  
   pio.h, [170](#)  
 ef\_pio\_unlink\_vi  
   pio.h, [170](#)  
 ef\_request\_id  
   ef\_vi.h, [135](#)  
 ef\_tcp\_checksum  
   checksum.h, [113](#)  
 ef\_tcp\_checksum\_ip6  
   checksum.h, [113](#)  
 ef\_tcp\_checksum\_ipx  
   checksum.h, [113](#)  
 ef\_udp\_checksum  
   checksum.h, [114](#)  
 ef\_udp\_checksum\_ip6  
   checksum.h, [114](#)  
 ef\_udp\_checksum\_ipx  
   checksum.h, [115](#)  
 ef\_vi, [68](#)  
   ef\_vi.h, [135](#)  
   ep\_state, [70](#)  
   evq\_base, [70](#)  
   evq\_mask, [70](#)  
   inited, [70](#)  
   io, [71](#)  
   linked\_pio, [71](#)  
   nic\_type, [71](#)  
   ops, [71](#)  
   rx\_buffer\_len, [71](#)  
   rx\_discard\_mask, [72](#)  
   rx\_prefix\_len, [72](#)  
   rx\_ts\_correction, [72](#)  
   timer\_quantum\_ns, [72](#)  
   ts\_format, [72](#)  
   tx\_alt\_hw2id, [73](#)  
   tx\_alt\_id2hw, [73](#)

- tx\_alt\_num, [73](#)
- tx\_push\_thresh, [73](#)
- tx\_ts\_correction\_ns, [73](#)
- vi\_clustered, [74](#)
- vi\_ctpio\_mmap\_ptr, [74](#)
- vi\_ctpio\_wb\_ticks, [74](#)
- vi\_flags, [74](#)
- vi\_i, [74](#)
- vi\_io\_mmap\_bytes, [75](#)
- vi\_io\_mmap\_ptr, [75](#)
- vi\_is\_normal, [75](#)
- vi\_is\_packed\_stream, [75](#)
- vi\_mem\_mmap\_bytes, [75](#)
- vi\_mem\_mmap\_ptr, [76](#)
- vi\_out\_flags, [76](#)
- vi\_ps\_buf\_size, [76](#)
- vi\_qs, [76](#)
- vi\_qs\_n, [76](#)
- vi\_resource\_id, [77](#)
- vi\_rxq, [77](#)
- vi\_stats, [77](#)
- vi\_txq, [77](#)
- ef\_vi.h, [115](#)
- EF\_EVENT\_RX\_MULTI\_CONT, [122](#)
- EF\_EVENT\_RX\_MULTI\_SOP, [123](#)
- EF\_EVENT\_RX\_PS\_NEXT\_BUFFER, [123](#)
- EF\_VI\_CTPIO\_CT\_THRESHOLD\_SNF, [124](#)
- ef\_eventq\_capacity, [139](#)
- ef\_eventq\_current, [139](#)
- ef\_eventq\_has\_event, [140](#)
- ef\_eventq\_has\_many\_events, [140](#)
- ef\_eventq\_poll, [123](#)
- ef\_eventq\_prime, [124](#)
- ef\_request\_id, [135](#)
- ef\_vi, [135](#)
- ef\_vi\_arch, [136](#)
- ef\_vi\_driver\_interface\_str, [140](#)
- ef\_vi\_flags, [137](#), [141](#)
- ef\_vi\_instance, [141](#)
- ef\_vi\_layout\_type, [138](#)
- ef\_vi\_out\_flags, [138](#)
- ef\_vi\_receive\_buffer\_len, [141](#)
- ef\_vi\_receive\_capacity, [142](#)
- ef\_vi\_receive\_fill\_level, [142](#)
- ef\_vi\_receive\_get\_bytes, [143](#)
- ef\_vi\_receive\_get\_timestamp, [143](#)
- ef\_vi\_receive\_get\_timestamp\_with\_sync\_flags, [144](#)
- ef\_vi\_receive\_init, [124](#)
- ef\_vi\_receive\_post, [145](#)
- ef\_vi\_receive\_prefix\_len, [145](#)
- ef\_vi\_receive\_push, [125](#)
- ef\_vi\_receive\_query\_layout, [146](#)
- ef\_vi\_receive\_set\_buffer\_len, [146](#)
- ef\_vi\_receive\_set\_discards, [147](#)
- ef\_vi\_receive\_space, [147](#)
- ef\_vi\_receive\_unbundle, [147](#)
- ef\_vi\_resource\_id, [148](#)
- ef\_vi\_rx\_discard\_err\_flags, [138](#)
- ef\_vi\_set\_tx\_push\_threshold, [148](#)
- ef\_vi\_transmit, [125](#)
- ef\_vi\_transmit\_alt\_discard, [126](#)
- ef\_vi\_transmit\_alt\_go, [126](#)
- ef\_vi\_transmit\_alt\_num\_ids, [149](#)
- ef\_vi\_transmit\_alt\_query\_overhead, [149](#)
- ef\_vi\_transmit\_alt\_select, [127](#)
- ef\_vi\_transmit\_alt\_select\_normal, [127](#)
- ef\_vi\_transmit\_alt\_stop, [128](#)
- ef\_vi\_transmit\_alt\_usage, [149](#)
- ef\_vi\_transmit\_capacity, [150](#)
- ef\_vi\_transmit\_copy\_pio, [128](#)
- ef\_vi\_transmit\_copy\_pio\_warm, [129](#)
- ef\_vi\_transmit\_ctpio, [150](#)
- ef\_vi\_transmit\_ctpio\_fallback, [151](#)
- ef\_vi\_transmit\_fill\_level, [152](#)
- ef\_vi\_transmit\_init, [152](#)
- ef\_vi\_transmit\_init\_undo, [152](#)
- ef\_vi\_transmit\_pio, [130](#)
- ef\_vi\_transmit\_pio\_warm, [131](#)
- ef\_vi\_transmit\_push, [131](#)
- ef\_vi\_transmit\_space, [153](#)
- ef\_vi\_transmit\_unbundle, [153](#)
- ef\_vi\_transmitv, [131](#)
- ef\_vi\_transmitv\_ctpio, [132](#)
- ef\_vi\_transmitv\_ctpio\_copy, [133](#)
- ef\_vi\_transmitv\_ctpio\_fallback, [154](#)
- ef\_vi\_transmitv\_init, [134](#)
- ef\_vi\_version\_str, [154](#)
- ef\_vi::ops, [93](#)
- eventq\_poll, [94](#)
- eventq\_prime, [94](#)
- eventq\_timer\_clear, [95](#)
- eventq\_timer\_prime, [95](#)
- eventq\_timer\_run, [95](#)
- eventq\_timer\_zero, [95](#)
- receive\_init, [95](#)
- receive\_push, [96](#)
- transmit, [96](#)
- transmit\_alt\_discard, [96](#)
- transmit\_alt\_go, [96](#)
- transmit\_alt\_select, [96](#)
- transmit\_alt\_select\_default, [97](#)
- transmit\_alt\_stop, [97](#)
- transmit\_copy\_pio, [97](#)
- transmit\_copy\_pio\_warm, [97](#)
- transmit\_pio, [97](#)
- transmit\_pio\_warm, [98](#)
- transmit\_push, [98](#)
- transmitv, [98](#)
- transmitv\_ctpio, [98](#)
- transmitv\_ctpio\_copy, [98](#)
- transmitv\_init, [99](#)

ef\_vi\_alloc\_from\_pd  
     vi.h, 188  
 ef\_vi\_alloc\_from\_set  
     vi.h, 189  
 ef\_vi\_arch  
     ef\_vi.h, 136  
 ef\_vi\_capabilities\_get  
     capabilities.h, 110  
 ef\_vi\_capabilities\_max  
     capabilities.h, 110  
 ef\_vi\_capabilities\_name  
     capabilities.h, 110  
 ef\_vi\_capability  
     capabilities.h, 108  
 ef\_vi\_driver\_interface\_str  
     ef\_vi.h, 140  
 ef\_vi\_filter\_add  
     vi.h, 191  
 ef\_vi\_filter\_del  
     vi.h, 191  
 ef\_vi\_flags  
     ef\_vi.h, 137, 141  
 ef\_vi\_flush  
     vi.h, 192  
 ef\_vi\_free  
     vi.h, 192  
 ef\_vi\_get\_mac  
     vi.h, 193  
 ef\_vi\_get\_pio\_size  
     pio.h, 171  
 ef\_vi\_instance  
     ef\_vi.h, 141  
 ef\_vi\_layout\_entry, 78  
     evle\_description, 78  
     evle\_offset, 78  
     evle\_type, 78  
 ef\_vi\_layout\_type  
     ef\_vi.h, 138  
 ef\_vi\_mtu  
     vi.h, 193  
 ef\_vi\_nic\_type, 79  
     arch, 79  
     nic\_flags, 79  
     revision, 80  
     variant, 80  
 ef\_vi\_out\_flags  
     ef\_vi.h, 138  
 ef\_vi\_pace  
     vi.h, 194  
 ef\_vi\_packed\_stream\_get\_params  
     packedstream.h, 161  
 ef\_vi\_packed\_stream\_unbundle  
     packedstream.h, 161  
 ef\_vi\_prime  
     vi.h, 194  
 ef\_vi\_receive\_buffer\_len  
     ef\_vi.h, 141  
 ef\_vi\_receive\_capacity  
     ef\_vi.h, 142  
 ef\_vi\_receive\_fill\_level  
     ef\_vi.h, 142  
 ef\_vi\_receive\_get\_bytes  
     ef\_vi.h, 143  
 ef\_vi\_receive\_get\_timestamp  
     ef\_vi.h, 143  
 ef\_vi\_receive\_get\_timestamp\_with\_sync\_flags  
     ef\_vi.h, 144  
 ef\_vi\_receive\_init  
     ef\_vi.h, 124  
 ef\_vi\_receive\_post  
     ef\_vi.h, 145  
 ef\_vi\_receive\_prefix\_len  
     ef\_vi.h, 145  
 ef\_vi\_receive\_push  
     ef\_vi.h, 125  
 ef\_vi\_receive\_query\_layout  
     ef\_vi.h, 146  
 ef\_vi\_receive\_set\_buffer\_len  
     ef\_vi.h, 146  
 ef\_vi\_receive\_set\_discards  
     ef\_vi.h, 147  
 ef\_vi\_receive\_space  
     ef\_vi.h, 147  
 ef\_vi\_receive\_unbundle  
     ef\_vi.h, 147  
 ef\_vi\_resource\_id  
     ef\_vi.h, 148  
 ef\_vi\_rx\_discard\_err\_flags  
     ef\_vi.h, 138  
 ef\_vi\_rxq, 80  
     descriptors, 81  
     ids, 81  
     mask, 81  
 ef\_vi\_rxq\_state, 82  
     added, 82  
     bytes\_acc, 82  
     in\_jumbo, 82  
     last\_desc\_i, 83  
     posted, 83  
     removed, 83  
     rx\_ps\_credit\_avail, 83  
 ef\_vi\_set, 84  
     vis\_pd, 84  
     vis\_res\_id, 84  
 ef\_vi\_set\_alloc\_from\_pd  
     vi.h, 195  
 ef\_vi\_set\_filter\_add  
     vi.h, 195  
 ef\_vi\_set\_filter\_del  
     vi.h, 196  
 ef\_vi\_set\_free  
     vi.h, 196

- ef\_vi\_set\_tx\_push\_threshold
  - ef\_vi.h, 148
- ef\_vi\_state, 85
  - evq, 85
  - rxq, 85
  - txq, 85
- ef\_vi\_stats, 86
  - evq\_gap, 86
  - rx\_ev\_bad\_desc\_i, 86
  - rx\_ev\_bad\_q\_label, 87
  - rx\_ev\_lost, 87
- ef\_vi\_stats\_field\_layout, 87
  - evsfl\_name, 88
  - evsfl\_offset, 88
  - evsfl\_size, 88
- ef\_vi\_stats\_layout, 88
  - evsl\_data\_size, 89
  - evsl\_fields, 89
  - evsl\_fields\_num, 89
- ef\_vi\_stats\_query
  - vi.h, 196
- ef\_vi\_stats\_query\_layout
  - vi.h, 197
- ef\_vi\_transmit
  - ef\_vi.h, 125
- ef\_vi\_transmit\_alt\_alloc
  - vi.h, 197
- ef\_vi\_transmit\_alt\_discard
  - ef\_vi.h, 126
- ef\_vi\_transmit\_alt\_free
  - vi.h, 198
- ef\_vi\_transmit\_alt\_go
  - ef\_vi.h, 126
- ef\_vi\_transmit\_alt\_num\_ids
  - ef\_vi.h, 149
- ef\_vi\_transmit\_alt\_overhead, 90
  - mask, 90
  - post\_round, 90
  - pre\_round, 90
- ef\_vi\_transmit\_alt\_query\_buffering
  - vi.h, 199
- ef\_vi\_transmit\_alt\_query\_overhead
  - ef\_vi.h, 149
- ef\_vi\_transmit\_alt\_select
  - ef\_vi.h, 127
- ef\_vi\_transmit\_alt\_select\_normal
  - ef\_vi.h, 127
- ef\_vi\_transmit\_alt\_stop
  - ef\_vi.h, 128
- ef\_vi\_transmit\_alt\_usage
  - ef\_vi.h, 149
- ef\_vi\_transmit\_capacity
  - ef\_vi.h, 150
- ef\_vi\_transmit\_copy\_pio
  - ef\_vi.h, 128
- ef\_vi\_transmit\_copy\_pio\_warm
  - ef\_vi.h, 129
- ef\_vi\_transmit\_ctpio
  - ef\_vi.h, 150
- ef\_vi\_transmit\_ctpio\_fallback
  - ef\_vi.h, 151
- ef\_vi\_transmit\_fill\_level
  - ef\_vi.h, 152
- ef\_vi\_transmit\_init
  - ef\_vi.h, 152
- ef\_vi\_transmit\_init\_undo
  - ef\_vi.h, 152
- ef\_vi\_transmit\_pio
  - ef\_vi.h, 130
- ef\_vi\_transmit\_pio\_warm
  - ef\_vi.h, 131
- ef\_vi\_transmit\_push
  - ef\_vi.h, 131
- ef\_vi\_transmit\_space
  - ef\_vi.h, 153
- ef\_vi\_transmit\_unbundle
  - ef\_vi.h, 153
- ef\_vi\_transmitv
  - ef\_vi.h, 131
- ef\_vi\_transmitv\_ctpio
  - ef\_vi.h, 132
- ef\_vi\_transmitv\_ctpio\_copy
  - ef\_vi.h, 133
- ef\_vi\_transmitv\_ctpio\_fallback
  - ef\_vi.h, 154
- ef\_vi\_transmitv\_init
  - ef\_vi.h, 134
- ef\_vi\_txq, 91
  - descriptors, 91
  - ids, 91
  - mask, 92
- ef\_vi\_txq\_state, 92
  - added, 92
  - previous, 93
  - removed, 93
  - ts\_nsec, 93
- ef\_vi\_version\_str
  - ef\_vi.h, 154
- EF\_EVENT\_RX\_DISCARD\_CRC\_BAD
  - ef\_vi.h, 136
- EF\_EVENT\_RX\_DISCARD\_CSUM\_BAD
  - ef\_vi.h, 136
- EF\_EVENT\_RX\_DISCARD\_EV\_ERROR
  - ef\_vi.h, 136
- EF\_EVENT\_RX\_DISCARD\_INNER\_CSUM\_BAD
  - ef\_vi.h, 136
- EF\_EVENT\_RX\_DISCARD\_MAX
  - ef\_vi.h, 136
- EF\_EVENT\_RX\_DISCARD\_MCAST\_MISMATCH
  - ef\_vi.h, 136
- EF\_EVENT\_RX\_DISCARD\_OTHER
  - ef\_vi.h, 136

- EF\_EVENT\_RX\_DISCARD\_RIGHTS  
 ef\_vi.h, [136](#)
- EF\_EVENT\_RX\_DISCARD\_TRUNC  
 ef\_vi.h, [136](#)
- EF\_EVENT\_TX\_ERROR\_2BIG  
 ef\_vi.h, [136](#)
- EF\_EVENT\_TX\_ERROR\_BUS  
 ef\_vi.h, [136](#)
- EF\_EVENT\_TX\_ERROR\_OFLOW  
 ef\_vi.h, [136](#)
- EF\_EVENT\_TX\_ERROR\_RIGHTS  
 ef\_vi.h, [136](#)
- EF\_EVENT\_TYPE\_OFLOW  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_RX  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_RX\_DISCARD  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_RX\_MULTI  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_RX\_NO\_DESC\_TRUNC  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_RX\_PACKED\_STREAM  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_SW  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_TX  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_TX\_ALT  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_TX\_ERROR  
 ef\_vi.h, [135](#)
- EF\_EVENT\_TYPE\_TX\_WITH\_TIMESTAMP  
 ef\_vi.h, [135](#)
- EF\_FILTER\_FLAG\_MCAST\_LOOP\_RECEIVE  
 vi.h, [178](#)
- EF\_FILTER\_FLAG\_NONE  
 vi.h, [178](#)
- EF\_FILTER\_VLAN\_ID\_ANY  
 vi.h, [178](#)
- EF\_PD\_DEFAULT  
 pd.h, [163](#)
- EF\_PD\_MCAST\_LOOP  
 pd.h, [163](#)
- EF\_PD\_MEMREG\_64KiB  
 pd.h, [163](#)
- EF\_PD\_PHYS\_MODE  
 pd.h, [163](#)
- EF\_PD\_RX\_PACKED\_STREAM  
 pd.h, [163](#)
- EF\_PD\_VF  
 pd.h, [163](#)
- EF\_PD\_VPORT  
 pd.h, [163](#)
- ef\_vi.h
- EF\_EVENT\_RX\_DISCARD\_CRC\_BAD, [136](#)
- EF\_EVENT\_RX\_DISCARD\_CSUM\_BAD, [136](#)
- EF\_EVENT\_RX\_DISCARD\_EV\_ERROR, [136](#)
- EF\_EVENT\_RX\_DISCARD\_INNER\_CSUM\_BAD,  
[136](#)
- EF\_EVENT\_RX\_DISCARD\_MAX, [136](#)
- EF\_EVENT\_RX\_DISCARD\_MCAST\_MISMATCH,  
[136](#)
- EF\_EVENT\_RX\_DISCARD\_OTHER, [136](#)
- EF\_EVENT\_RX\_DISCARD\_RIGHTS, [136](#)
- EF\_EVENT\_RX\_DISCARD\_TRUNC, [136](#)
- EF\_EVENT\_TX\_ERROR\_2BIG, [136](#)
- EF\_EVENT\_TX\_ERROR\_BUS, [136](#)
- EF\_EVENT\_TX\_ERROR\_OFLOW, [136](#)
- EF\_EVENT\_TX\_ERROR\_RIGHTS, [136](#)
- EF\_EVENT\_TYPE\_OFLOW, [135](#)
- EF\_EVENT\_TYPE\_RX, [135](#)
- EF\_EVENT\_TYPE\_RX\_DISCARD, [135](#)
- EF\_EVENT\_TYPE\_RX\_MULTI, [135](#)
- EF\_EVENT\_TYPE\_RX\_MULTI\_DISCARD, [135](#)
- EF\_EVENT\_TYPE\_RX\_NO\_DESC\_TRUNC, [135](#)
- EF\_EVENT\_TYPE\_RX\_PACKED\_STREAM, [135](#)
- EF\_EVENT\_TYPE\_SW, [135](#)
- EF\_EVENT\_TYPE\_TX, [135](#)
- EF\_EVENT\_TYPE\_TX\_ALT, [135](#)
- EF\_EVENT\_TYPE\_TX\_ERROR, [135](#)
- EF\_EVENT\_TYPE\_TX\_WITH\_TIMESTAMP, [135](#)
- EF\_VI\_ARCH\_EF10, [137](#)
- EF\_VI\_ARCH\_FALCON, [137](#)
- EF\_VI\_DISCARD\_RX\_ETH\_FCS\_ERR, [139](#)
- EF\_VI\_DISCARD\_RX\_ETH\_LEN\_ERR, [139](#)
- EF\_VI\_DISCARD\_RX\_L3\_CSUM\_ERR, [139](#)
- EF\_VI\_DISCARD\_RX\_L4\_CSUM\_ERR, [139](#)
- EF\_VI\_DISCARD\_RX\_TOBE\_DISC, [139](#)
- EF\_VI\_ENABLE\_EV\_TIMER, [138](#)
- EF\_VI\_FLAGS\_DEFAULT, [137](#)
- EF\_VI\_ISCSI\_RX\_DDIG, [137](#)
- EF\_VI\_ISCSI\_RX\_HDIG, [137](#)
- EF\_VI\_ISCSI\_TX\_DDIG, [137](#)
- EF\_VI\_ISCSI\_TX\_HDIG, [137](#)
- EF\_VI\_LAYOUT\_FRAME, [138](#)
- EF\_VI\_LAYOUT\_MINOR\_TICKS, [138](#)
- EF\_VI\_LAYOUT\_PACKET\_LENGTH, [138](#)
- EF\_VI\_OUT\_CLOCK\_SYNC\_STATUS, [138](#)
- EF\_VI\_RX\_EVENT\_MERGE, [138](#)
- EF\_VI\_RX\_PACKED\_STREAM, [137](#)
- EF\_VI\_RX\_PHYS\_ADDR, [137](#)
- EF\_VI\_RX\_PS\_BUF\_SIZE\_64K, [138](#)
- EF\_VI\_RX\_TIMESTAMPS, [137](#)
- EF\_VI\_TX\_ALT, [138](#)
- EF\_VI\_TX\_CTPIO, [138](#)
- EF\_VI\_TX\_CTPIO\_NO\_POISON, [138](#)
- EF\_VI\_TX\_FILTER\_IP, [137](#)
- EF\_VI\_TX\_FILTER\_MAC, [137](#)
- EF\_VI\_TX\_FILTER\_MASK\_1, [137](#)

- EF\_VI\_TX\_FILTER\_MASK\_2, [137](#)
- EF\_VI\_TX\_FILTER\_MASK\_3, [137](#)
- EF\_VI\_TX\_IP\_CSUM\_DIS, [137](#)
- EF\_VI\_TX\_PHYS\_ADDR, [137](#)
- EF\_VI\_TX\_PUSH\_ALWAYS, [137](#)
- EF\_VI\_TX\_PUSH\_DISABLE, [137](#)
- EF\_VI\_TX\_TCPUDP\_CSUM\_DIS, [137](#)
- EF\_VI\_TX\_TCPUDP\_ONLY, [137](#)
- EF\_VI\_TX\_TIMESTAMPS, [137](#)
- EF\_VI\_ARCH\_EF10
  - [ef\\_vi.h, 137](#)
- EF\_VI\_ARCH\_FALCON
  - [ef\\_vi.h, 137](#)
- EF\_VI\_CAP\_BUFFER\_MODE
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_CTPIO
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_EVQ\_SIZES
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_HW\_MULTICAST\_LOOPBACK
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_HW\_MULTICAST\_REPLICATION
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_HW\_RX\_TIMESTAMPING
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_HW\_TX\_TIMESTAMPING
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_MAC\_SPOOFING
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_MAX
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_MULTICAST\_FILTER\_CHAINING
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_NIC\_PACE
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_PACKED\_STREAM
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_PACKED\_STREAM\_BUFFER\_SIZES
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_PHYS\_MODE
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_PIO
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_PIO\_BUFFER\_COUNT
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_PIO\_BUFFER\_SIZE
  - [capabilities.h, 108](#)
- EF\_VI\_CAP\_RX\_FILTER\_ETHERTYPE
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_IP4\_PROTO
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_ETH\_LOCAL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_ETH\_LOCAL\_VLAN
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_IP6\_VLAN
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_IP\_VLAN
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_MCAST\_ALL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_MCAST\_MISMATCH
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_SNIFF
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP6\_FULL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP6\_LOCAL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP\_FULL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_TCP\_LOCAL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UCAST\_ALL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UCAST\_MISMATCH
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP6\_FULL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP6\_LOCAL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP\_FULL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FILTER\_TYPE\_UDP\_LOCAL
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FORCE\_EVENT\_MERGING
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_FW\_VARIANT
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RX\_MERGE
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_RXQ\_SIZES
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_ALTERNATIVES
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_ALTERNATIVES\_CP\_BUFFER\_SIZE
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_ALTERNATIVES\_CP\_BUFFERS
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_ALTERNATIVES\_VFIFOS
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_FILTER\_TYPE\_SNIFF
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_FW\_VARIANT
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TX\_PUSH\_ALWAYS
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_TXQ\_SIZES
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_VPORTS
  - [capabilities.h, 109](#)
- EF\_VI\_CAP\_ZERO\_RX\_PREFIX

[capabilities.h, 109](#)  
[EF\\_VI\\_DISCARD\\_RX\\_ETH\\_FCS\\_ERR](#)  
[ef\\_vi.h, 139](#)  
[EF\\_VI\\_DISCARD\\_RX\\_ETH\\_LEN\\_ERR](#)  
[ef\\_vi.h, 139](#)  
[EF\\_VI\\_DISCARD\\_RX\\_L3\\_CSUM\\_ERR](#)  
[ef\\_vi.h, 139](#)  
[EF\\_VI\\_DISCARD\\_RX\\_L4\\_CSUM\\_ERR](#)  
[ef\\_vi.h, 139](#)  
[EF\\_VI\\_DISCARD\\_RX\\_TOBE\\_DISC](#)  
[ef\\_vi.h, 139](#)  
[EF\\_VI\\_ENABLE\\_EV\\_TIMER](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_FLAGS\\_DEFAULT](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_ISCSI\\_RX\\_DDIG](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_ISCSI\\_RX\\_HDIG](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_ISCSI\\_TX\\_DDIG](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_ISCSI\\_TX\\_HDIG](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_LAYOUT\\_FRAME](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_LAYOUT\\_MINOR\\_TICKS](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_LAYOUT\\_PACKET\\_LENGTH](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_OUT\\_CLOCK\\_SYNC\\_STATUS](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_RX\\_EVENT\\_MERGE](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_RX\\_PACKED\\_STREAM](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_RX\\_PHYS\\_ADDR](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_RX\\_PS\\_BUF\\_SIZE\\_64K](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_RX\\_TIMESTAMPS](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_ALT](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_TX\\_CTIPIO](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_TX\\_CTIPIO\\_NO\\_POISON](#)  
[ef\\_vi.h, 138](#)  
[EF\\_VI\\_TX\\_FILTER\\_IP](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_FILTER\\_MAC](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_FILTER\\_MASK\\_1](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_FILTER\\_MASK\\_2](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_FILTER\\_MASK\\_3](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_IP\\_CSUM\\_DIS](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_PHYS\\_ADDR](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_PUSH\\_ALWAYS](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_PUSH\\_DISABLE](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_TCPUDP\\_CSUM\\_DIS](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_TCPUDP\\_ONLY](#)  
[ef\\_vi.h, 137](#)  
[EF\\_VI\\_TX\\_TIMESTAMPS](#)  
[ef\\_vi.h, 137](#)  
[ep\\_state](#)  
[ef\\_vi, 70](#)  
[eventq\\_poll](#)  
[ef\\_vi::ops, 94](#)  
[eventq\\_prime](#)  
[ef\\_vi::ops, 94](#)  
[eventq\\_timer\\_clear](#)  
[ef\\_vi::ops, 95](#)  
[eventq\\_timer\\_prime](#)  
[ef\\_vi::ops, 95](#)  
[eventq\\_timer\\_run](#)  
[ef\\_vi::ops, 95](#)  
[eventq\\_timer\\_zero](#)  
[ef\\_vi::ops, 95](#)  
[evle\\_description](#)  
[ef\\_vi\\_layout\\_entry, 78](#)  
[evle\\_offset](#)  
[ef\\_vi\\_layout\\_entry, 78](#)  
[evle\\_type](#)  
[ef\\_vi\\_layout\\_entry, 78](#)  
[evq](#)  
[ef\\_vi\\_state, 85](#)  
[evq\\_base](#)  
[ef\\_vi, 70](#)  
[evq\\_clear\\_stride](#)  
[ef\\_eventq\\_state, 55](#)  
[evq\\_gap](#)  
[ef\\_vi\\_stats, 86](#)  
[evq\\_mask](#)  
[ef\\_vi, 70](#)  
[evq\\_ptr](#)  
[ef\\_eventq\\_state, 55](#)  
[evsfl\\_name](#)  
[ef\\_vi\\_stats\\_field\\_layout, 88](#)  
[evsfl\\_offset](#)  
[ef\\_vi\\_stats\\_field\\_layout, 88](#)  
[evsfl\\_size](#)  
[ef\\_vi\\_stats\\_field\\_layout, 88](#)  
[evsl\\_data\\_size](#)  
[ef\\_vi\\_stats\\_layout, 89](#)  
[evsl\\_fields](#)

- ef\_vi\_stats\_layout, 89
- evsl\_fields\_num
  - ef\_vi\_stats\_layout, 89
- filter\_id
  - ef\_filter\_cookie, 57
- filter\_type
  - ef\_filter\_cookie, 57
- flags
  - ef\_filter\_spec, 58
- generic
  - ef\_event, 53
- ids
  - ef\_vi\_rxq, 81
  - ef\_vi\_txq, 91
- in\_jumbo
  - ef\_vi\_rxq\_state, 82
- inited
  - ef\_vi, 70
- io
  - ef\_vi, 71
- iov\_len
  - ef\_iovec, 60
- last\_desc\_i
  - ef\_vi\_rxq\_state, 83
- linked\_pio
  - ef\_vi, 71
- mask
  - ef\_vi\_rxq, 81
  - ef\_vi\_transmit\_alt\_overhead, 90
  - ef\_vi\_txq, 92
- memreg.h, 155
  - ef\_memreg\_alloc, 156
  - ef\_memreg\_dma\_addr, 157
  - ef\_memreg\_free, 157
- mr\_dma\_addrs
  - ef\_memreg, 61
- mr\_dma\_addrs\_base
  - ef\_memreg, 61
- nic\_flags
  - ef\_vi\_nic\_type, 79
- nic\_type
  - ef\_vi, 71
- ops
  - ef\_vi, 71
- packedstream.h, 158
  - EF\_VI\_PS\_FLAG\_BAD\_IP\_CSUM, 159
  - ef\_packed\_stream\_packet\_first, 160
  - ef\_packed\_stream\_packet\_next, 160
  - ef\_packed\_stream\_packet\_payload, 160
  - ef\_vi\_packed\_stream\_get\_params, 161
  - ef\_vi\_packed\_stream\_unbundle, 161
- pd.h, 162
  - ef\_pd\_alloc, 164
  - ef\_pd\_alloc\_by\_name, 164
  - ef\_pd\_alloc\_with\_vport, 165
  - ef\_pd\_flags, 163
  - ef\_pd\_free, 165
  - ef\_pd\_interface\_name, 166
- pd.h
  - EF\_PD\_DEFAULT, 163
  - EF\_PD\_MCAST\_LOOP, 163
  - EF\_PD\_MEMREG\_64KiB, 163
  - EF\_PD\_PHYS\_MODE, 163
  - EF\_PD\_RX\_PACKED\_STREAM, 163
  - EF\_PD\_VF, 163
  - EF\_PD\_VPORT, 163
- pd\_cluster\_dh
  - ef\_pd, 65
- pd\_cluster\_name
  - ef\_pd, 65
- pd\_cluster\_sock
  - ef\_pd, 66
- pd\_cluster\_viset\_index
  - ef\_pd, 66
- pd\_cluster\_viset\_resource\_id
  - ef\_pd, 66
- pd\_flags
  - ef\_pd, 66
- pd\_intf\_name
  - ef\_pd, 66
- pd\_resource\_id
  - ef\_pd, 67
- pio.h, 166
  - ef\_pio\_alloc, 167
  - ef\_pio\_free, 169
  - ef\_pio\_link\_vi, 169
  - ef\_pio\_memcpy, 170
  - ef\_pio\_unlink\_vi, 170
  - ef\_vi\_get\_pio\_size, 171
- pio\_buffer
  - ef\_pio, 67
- pio\_io
  - ef\_pio, 68
- pio\_len
  - ef\_pio, 68
- pio\_resource\_id
  - ef\_pio, 68
- post\_round
  - ef\_vi\_transmit\_alt\_overhead, 90
- posted
  - ef\_vi\_rxq\_state, 83
- pre\_round
  - ef\_vi\_transmit\_alt\_overhead, 90
- previous
  - ef\_vi\_txq\_state, 93
- ps\_cap\_len



- ef\_packed\_stream\_packet, [62](#)
- ps\_flags
  - ef\_packed\_stream\_packet, [62](#)
- ps\_next\_offset
  - ef\_packed\_stream\_packet, [62](#)
- ps\_orig\_len
  - ef\_packed\_stream\_packet, [62](#)
- ps\_pkt\_start\_offset
  - ef\_packed\_stream\_packet, [62](#)
- ps\_ts\_nsec
  - ef\_packed\_stream\_packet, [63](#)
- ps\_ts\_sec
  - ef\_packed\_stream\_packet, [63](#)
- psp\_buffer\_align
  - ef\_packed\_stream\_params, [64](#)
- psp\_buffer\_size
  - ef\_packed\_stream\_params, [64](#)
- psp\_max\_usable\_buffers
  - ef\_packed\_stream\_params, [64](#)
- psp\_start\_offset
  - ef\_packed\_stream\_params, [64](#)
- receive\_init
  - ef\_vi::ops, [95](#)
- receive\_push
  - ef\_vi::ops, [96](#)
- removed
  - ef\_vi\_rxq\_state, [83](#)
  - ef\_vi\_txq\_state, [93](#)
- revision
  - ef\_vi\_nic\_type, [80](#)
- rx
  - ef\_event, [53](#)
- rx\_buffer\_len
  - ef\_vi, [71](#)
- rx\_discard
  - ef\_event, [53](#)
- rx\_discard\_mask
  - ef\_vi, [72](#)
- rx\_ev\_bad\_desc\_i
  - ef\_vi\_stats, [86](#)
- rx\_ev\_bad\_q\_label
  - ef\_vi\_stats, [87](#)
- rx\_ev\_lost
  - ef\_vi\_stats, [87](#)
- rx\_multi
  - ef\_event, [53](#)
- rx\_multi\_discard
  - ef\_event, [53](#)
- rx\_no\_desc\_trunc
  - ef\_event, [54](#)
- rx\_packed\_stream
  - ef\_event, [54](#)
- rx\_prefix\_len
  - ef\_vi, [72](#)
- rx\_ps\_credit\_avail
  - ef\_vi\_rxq\_state, [83](#)
- rx\_ts\_correction
  - ef\_vi, [72](#)
- rxq
  - ef\_vi\_state, [85](#)
- sw
  - ef\_event, [54](#)
- sync\_flags
  - ef\_eventq\_state, [56](#)
- sync\_timestamp\_major
  - ef\_eventq\_state, [56](#)
- sync\_timestamp\_minimum
  - ef\_eventq\_state, [56](#)
- sync\_timestamp\_minor
  - ef\_eventq\_state, [56](#)
- sync\_timestamp\_synchronised
  - ef\_eventq\_state, [56](#)
- timer.h, [171](#)
  - ef\_eventq\_timer\_clear, [172](#)
  - ef\_eventq\_timer\_prime, [172](#)
  - ef\_eventq\_timer\_run, [173](#)
  - ef\_eventq\_timer\_zero, [174](#)
- timer\_quantum\_ns
  - ef\_vi, [72](#)
- transmit
  - ef\_vi::ops, [96](#)
- transmit\_alt\_discard
  - ef\_vi::ops, [96](#)
- transmit\_alt\_go
  - ef\_vi::ops, [96](#)
- transmit\_alt\_select
  - ef\_vi::ops, [96](#)
- transmit\_alt\_select\_default
  - ef\_vi::ops, [97](#)
- transmit\_alt\_stop
  - ef\_vi::ops, [97](#)
- transmit\_copy\_pio
  - ef\_vi::ops, [97](#)
- transmit\_copy\_pio\_warm
  - ef\_vi::ops, [97](#)
- transmit\_pio
  - ef\_vi::ops, [97](#)
- transmit\_pio\_warm
  - ef\_vi::ops, [98](#)
- transmit\_push
  - ef\_vi::ops, [98](#)
- transmitv
  - ef\_vi::ops, [98](#)
- transmitv\_ctpio
  - ef\_vi::ops, [98](#)
- transmitv\_ctpio\_copy
  - ef\_vi::ops, [98](#)
- transmitv\_init
  - ef\_vi::ops, [99](#)
- ts\_format

- ef\_vi, 72
- ts\_nsec
  - ef\_vi\_txq\_state, 93
- tx
  - ef\_event, 54
- tx\_alt
  - ef\_event, 54
- tx\_alt\_hw2id
  - ef\_vi, 73
- tx\_alt\_id2hw
  - ef\_vi, 73
- tx\_alt\_num
  - ef\_vi, 73
- tx\_error
  - ef\_event, 54
- tx\_push\_thresh
  - ef\_vi, 73
- tx\_timestamp
  - ef\_event, 54
- tx\_ts\_correction\_ns
  - ef\_vi, 73
- txq
  - ef\_vi\_state, 85
- type
  - ef\_filter\_spec, 59
- variant
  - ef\_vi\_nic\_type, 80
- vi.h, 175
  - ef\_eventq\_put, 178
  - ef\_filter\_flags, 178
  - ef\_filter\_spec\_init, 178
  - ef\_filter\_spec\_set\_block\_kernel, 180
  - ef\_filter\_spec\_set\_block\_kernel\_multicast, 180
  - ef\_filter\_spec\_set\_block\_kernel\_unicast, 181
  - ef\_filter\_spec\_set\_eth\_local, 181
  - ef\_filter\_spec\_set\_eth\_type, 181
  - ef\_filter\_spec\_set\_ip4\_full, 182
  - ef\_filter\_spec\_set\_ip4\_local, 183
  - ef\_filter\_spec\_set\_ip6\_full, 183
  - ef\_filter\_spec\_set\_ip6\_local, 184
  - ef\_filter\_spec\_set\_ip\_proto, 184
  - ef\_filter\_spec\_set\_multicast\_all, 185
  - ef\_filter\_spec\_set\_multicast\_mismatch, 185
  - ef\_filter\_spec\_set\_port\_sniff, 186
  - ef\_filter\_spec\_set\_tx\_port\_sniff, 186
  - ef\_filter\_spec\_set\_unicast\_all, 187
  - ef\_filter\_spec\_set\_unicast\_mismatch, 187
  - ef\_filter\_spec\_set\_vlan, 187
  - ef\_vi\_alloc\_from\_pd, 188
  - ef\_vi\_alloc\_from\_set, 189
  - ef\_vi\_filter\_add, 191
  - ef\_vi\_filter\_del, 191
  - ef\_vi\_flush, 192
  - ef\_vi\_free, 192
  - ef\_vi\_get\_mac, 193
  - ef\_vi\_mtu, 193
  - ef\_vi\_pace, 194
  - ef\_vi\_prime, 194
  - ef\_vi\_set\_alloc\_from\_pd, 195
  - ef\_vi\_set\_filter\_add, 195
  - ef\_vi\_set\_filter\_del, 196
  - ef\_vi\_set\_free, 196
  - ef\_vi\_stats\_query, 196
  - ef\_vi\_stats\_query\_layout, 197
  - ef\_vi\_transmit\_alt\_alloc, 197
  - ef\_vi\_transmit\_alt\_free, 198
  - ef\_vi\_transmit\_alt\_query\_buffering, 199
- vi.h
  - EF\_FILTER\_FLAG\_MCAST\_LOOP\_RECEIVE, 178
  - EF\_FILTER\_FLAG\_NONE, 178
  - EF\_FILTER\_VLAN\_ID\_ANY, 178
- vi\_clustered
  - ef\_vi, 74
- vi\_ctpio\_mmap\_ptr
  - ef\_vi, 74
- vi\_ctpio\_wb\_ticks
  - ef\_vi, 74
- vi\_flags
  - ef\_vi, 74
- vi\_j
  - ef\_vi, 74
- vi\_io\_mmap\_bytes
  - ef\_vi, 75
- vi\_io\_mmap\_ptr
  - ef\_vi, 75
- vi\_is\_normal
  - ef\_vi, 75
- vi\_is\_packed\_stream
  - ef\_vi, 75
- vi\_mem\_mmap\_bytes
  - ef\_vi, 75
- vi\_mem\_mmap\_ptr
  - ef\_vi, 76
- vi\_out\_flags
  - ef\_vi, 76
- vi\_ps\_buf\_size
  - ef\_vi, 76
- vi\_qs
  - ef\_vi, 76
- vi\_qs\_n
  - ef\_vi, 76
- vi\_resource\_id
  - ef\_vi, 77
- vi\_rxq
  - ef\_vi, 77
- vi\_stats
  - ef\_vi, 77
- vi\_txq
  - ef\_vi, 77
- vis\_pd

ef\_vi\_set, 84  
vis\_res\_id  
ef\_vi\_set, 84