ESP8266 Non-OS SDK

API Reference



About This Guide

This document lists ESP8266_NONOS_SDK APIs.

Release Notes

Date	Version	Release notes
2016.01	V1.5.2	First Release
2016.03	V1.5.2	Updated Section 3.2, 9.5 and 3.3.37
2016.04	V1.5.3	Added Section 3.5.11 and 3.5.12
2010.04	V 1.0.0	Updated Section 3.5.67 and 3.7.9
		Added Section 3.3.8 and 3.7.8
2016.06	V1.5.4	Added Section 3.3.46, 3.3.47, and 3.3.48
		Updated Section 3.7
2016.07	V2.0.0	Added Section 3.9, 3.14, 3.3.48, 3.5.72. and 3.5.73
2010.01	V2.0.0	Updated Section 3.8.6 and 3.5.65
2016.11	V2.0.1	Changed prototype wifi_station_get_hostname to wifi_station_set_hostname in Section 3.5.30
2017.01	V2.0.2	Updated Chapter 2
2017.05	V2.1.0	Added Section 3.3.49, 3.3.50, 4.3.6 and 8.2.4
2017.05	V2.1.1	Updated Chapter 2
2017.06	V2.1.2	Updated Section 3.3.9
0010.00	\ /O. O.	Updated Section 6.2.1, 6.2.3, 6.2.4, 3.3.49
2018.02	V2.2	Added Section 3.4.8, 3.4.9, 3.5.74, 3.5.75
2018.05	V2.2.1	Updated Section 2.4, 3.5.54, 3.7
		Added Section 3.3.50, 3.3.51, 3.3.52, 3.5.76, 3.5.77, 3.5.78, 3.5.79, A.6
2018.08	V3.0	Updated Section 2.5
		Remove system_phy_freq_trace_enable

Documentation Change Notification

Espressif provides email notifications to keep customers updated on changes to technical documentation. Please subscribe at https://www.espressif.com/en/subscribe.

Certification

Download certificates for Espressif products from <u>https://www.espressif.com/en/certificates.</u>

Table of Contents

1.	Preambles		1
2.	Non-OS SDK		2
	2.1. Non-OS S	DK Introduction	2
	2.2. Code Struc	cture	2
	2.3. Timer and	Interrupt	4
		erformance	
	2.5. System Me	emory	4
3.	Application Pro	ogramming Interface (APIs)	7
		Гimer	
		mer_arm	
		mer_disarm	
		mer_setfn	
	3.1.4. syste	em_timer_reinit	8
	3.1.5. os_tii	mer_arm_us	8
	3.2. Hardware	Timer	9
	3.2.1. hw_ti	imer_init	9
	3.2.2. hw_ti	imer_arm	9
	3.2.3. hw_ti	imer_set_func	10
	3.2.4. Hard	lware Timer Example	10
	3.3. System AF	Pls	11
	3.3.1. syste	em_get_sdk_version	11
	3.3.2. syste	em_restore	11
	3.3.3. syste	em_restart	11
	3.3.4. syste	em_init_done_cb	11
	3.3.5. syste	em_get_chip_id	12
	3.3.6. syste	em_get_vdd33	12
	3.3.7. syste	em_adc_read	12
	3.3.8. syste	em_adc_read_fast	13
	3.3.9. syste	em_deep_sleep	14
	3.3.10. syste	em_deep_sleep_set_option	15
	3.3.11. syste	em_phy_set_rfoption	16
	3.3.12. svste	em_phy_set_powerup_option	16

3.3.13.	system_phy_set_max_tpw	17
3.3.14.	system_phy_set_tpw_via_vdd33	17
3.3.15.	system_set_os_print	17
3.3.16.	system_print_meminfo	17
3.3.17.	system_get_free_heap_size	18
3.3.18.	system_os_task	18
3.3.19.	system_os_post	19
3.3.20.	system_get_time	19
3.3.21.	system_get_rtc_time	19
3.3.22.	system_rtc_clock_cali_proc	20
3.3.23.	system_rtc_mem_write	20
3.3.24.	system_rtc_mem_read	21
3.3.25.	system_uart_swap	21
3.3.26.	system_uart_de_swap	21
3.3.27.	system_get_boot_version	22
3.3.28.	system_get_userbin_addr	22
3.3.29.	system_get_boot_mode	22
3.3.30.	system_restart_enhance	22
3.3.31.	system_update_cpu_freq	23
3.3.32.	system_get_cpu_freq	23
3.3.33.	system_get_flash_size_map	23
3.3.34.	system_get_rst_info	25
3.3.35.	system_soft_wdt_stop	25
3.3.36.	system_soft_wdt_restart	26
3.3.37.	system_soft_wdt_feed	26
3.3.38.	system_show_malloc	26
3.3.39.	os_memset	27
3.3.40.	os_memcpy	27
3.3.41.	os_strlen	27
3.3.42.	os_printf	27
3.3.43.	os_bzero	28
3.3.44.	os_delay_us	28
3.3.45.	os_install_putc1	28
3.3.46.	os_random	28
3.3.47.	os_get_random	29
3.3.48.	user_rf_cal_sector_set	29
3.3.49.	system_deep_sleep_instant	30

	3.3.50.	system_partition_table_regist	31
	3.3.51.	system_partition_get_ota_partition_size	31
	3.3.52.	system_partition_get_item	32
3.4	I. SPIF	Flash Related APIs	32
	3.4.1.	spi_flash_get_id	32
	3.4.2.	spi_flash_erase_sector	32
	3.4.3.	spi_flash_write	32
	3.4.4.	spi_flash_read	33
	3.4.5.	system_param_save_with_protect	33
	3.4.6.	system_param_load	34
	3.4.7.	spi_flash_set_read_func	35
	3.4.8.	spi_flash_erase_protect_enable	35
	3.4.9.	spi_flash_erase_protect_disable	35
3.5	. Wi-F	i Related APIs	36
	3.5.1.	wifi_get_opmode	36
	3.5.2.	wifi_get_opmode_default	36
	3.5.3.	wifi_set_opmode	37
	3.5.4.	wifi_set_opmode_current	37
	3.5.5.	wifi_station_get_config	37
	3.5.6.	wifi_station_get_config_default	38
	3.5.7.	wifi_station_set_config	38
	3.5.8.	wifi_station_set_config_current	39
	3.5.9.	wifi_station_set_cert_key	39
	3.5.10.	wifi_station_clear_cert_key	40
	3.5.11.	wifi_station_set_username	41
	3.5.12.	wifi_station_clear_username	41
	3.5.13.	wifi_station_connect	41
	3.5.14.	wifi_station_disconnect	41
	3.5.15.	wifi_station_get_connect_status	42
	3.5.16.	wifi_station_scan	42
	3.5.17.	scan_done_cb_t	43
	3.5.18.	wifi_station_ap_number_set	43
	3.5.19.	wifi_station_get_ap_info	43
	3.5.20.	wifi_station_ap_change	44
	3.5.21.	wifi_station_get_current_ap_id	44
	3.5.22.	wifi_station_get_auto_connect	44
	3.5.23.	wifi_station_set_auto_connect	44

3.5.24.	wifi_station_dhcpc_start	.45
3.5.25.	wifi_station_dhcpc_stop	.45
3.5.26.	wifi_station_dhcpc_status	.45
3.5.27.	wifi_station_dhcpc_set_maxtry	.45
3.5.28.	wifi_station_set_reconnect_policy	.46
3.5.29.	wifi_station_get_rssi	.46
3.5.30.	wifi_station_set_hostname	.46
3.5.31.	wifi_station_get_hostname	.47
3.5.32.	wifi_softap_get_config	.47
3.5.33.	wifi_softap_get_config_default	.47
3.5.34.	wifi_softap_set_config	.47
3.5.35.	wifi_softap_set_config_current	.48
3.5.36.	wifi_softap_get_station_num	.48
3.5.37.	wifi_softap_get_station_info	.48
3.5.38.	wifi_softap_free_station_info	.48
3.5.39.	wifi_softap_dhcps_start	.49
3.5.40.	wifi_softap_dhcps_stop	.49
3.5.41.	wifi_softap_set_dhcps_lease	.50
3.5.42.	wifi_softap_get_dhcps_lease	.51
3.5.43.	wifi_softap_set_dhcps_lease_time	.51
3.5.44.	wifi_softap_get_dhcps_lease_time	.51
3.5.45.	wifi_softap_reset_dhcps_lease_time	.51
3.5.46.	wifi_softap_dhcps_status	.52
3.5.47.	wifi_softap_set_dhcps_offer_option	.52
3.5.48.	wifi_set_phy_mode	.52
3.5.49.	wifi_get_phy_mode	.53
3.5.50.	wifi_get_ip_info	.53
3.5.51.	wifi_set_ip_info	.53
3.5.52.	wifi_set_macaddr	.54
3.5.53.	wifi_get_macaddr	.55
3.5.54.	wifi_set_sleep_type	.55
3.5.55.	wifi_get_sleep_type	.56
3.5.56.	wifi_status_led_install	.56
3.5.57.	wifi_status_led_uninstall	.56
3.5.58.	wifi_set_broadcast_if	.57
3.5.59.	wifi_get_broadcast_if	.57
3.5.60.	wifi_set_event_handler_cb	.58

3.5.61.	wifi_wps_enable	59
3.5.62.	wifi_wps_disable	59
3.5.63.	wifi_wps_start	59
3.5.64.	wifi_set_wps_cb	60
3.5.65.	wifi_register_send_pkt_freedom_cb	61
3.5.66.	wifi_unregister_send_pkt_freedom_cb	61
3.5.67.	wifi_send_pkt_freedom	62
3.5.68.	wifi_rfid_locp_recv_open	62
3.5.69.	wifi_rfid_locp_recv_close	62
3.5.70.	wifi_register_rfid_locp_recv_cb	63
3.5.71.	wifi_unregister_rfid_locp_recv_cb	63
3.5.72.	wifi_enable_gpio_wakeup	63
3.5.73.	wifi_disable_gpio_wakeup	64
3.5.74.	wifi_set_country	64
3.5.75.	wifi_get_country	64
3.5.76.	wifi_set_sleep_level	65
3.5.77.	wifi_get_sleep_level	65
3.5.78.	wifi_set_listen_interval	65
3.5.79.	wifi_get_listen_interval	66
3.6. Rate	Control APIs	67
3.6.1.	wifi_set_user_fixed_rate	67
3.6.2.	wifi_get_user_fixed_rate	68
3.6.3.	wifi_set_user_sup_rate	68
3.6.4.	wifi_set_user_rate_limit	69
3.6.5.	wifi_set_user_limit_rate_mask	70
3.6.6.	wifi_get_user_limit_rate_mask	70
3.7. Force	e Sleep APIs	71
3.7.1.	wifi_fpm_open	71
3.7.2.	wifi_fpm_close	71
3.7.3.	wifi_fpm_do_wakeup	71
3.7.4.	wifi_fpm_set_wakeup_cb	72
3.7.5.	wifi_fpm_do_sleep	72
3.7.6.	wifi_fpm_set_sleep_type	72
3.7.7.	wifi_fpm_get_sleep_type	
3.7.8.	wifi_fpm_auto_sleep_set_in_null_mode	
3.7.9.	Example	
3.8. ESP-	NOW APIs	76

3.8.1.	Roles of ESP-NOW	76
3.8.2.	esp_now_init	76
3.8.3.	esp_now_deinit	77
3.8.4.	esp_now_register_recv_cb	77
3.8.5.	esp_now_unregister_recv_cb	77
3.8.6.	esp_now_register_send_cb	77
3.8.7.	esp_now_unregister_send_cb	79
3.8.8.	esp_now_send	79
3.8.9.	esp_now_add_peer	79
3.8.10.	esp_now_del_peer	80
3.8.11.	esp_now_set_self_role	80
3.8.12.	esp_now_get_self_role	80
3.8.13.	esp_now_set_peer_role	80
3.8.14.	esp_now_get_peer_role	81
3.8.15.	esp_now_set_peer_key	81
3.8.16.	esp_now_get_peer_key	81
3.8.17.	esp_now_set_peer_channel	81
3.8.18.	esp_now_get_peer_channel	82
3.8.19.	esp_now_is_peer_exist	82
3.8.20.	esp_now_fetch_peer	82
3.8.21.	esp_now_get_cnt_info	83
3.8.22.	esp_now_set_kok	83
3.9. Simp	ple Pair APIs	84
3.9.1.	Status of Simple Pair	84
3.9.2.	register_simple_pair_status_cb	84
3.9.3.	unregister_simple_pair_status_cb	84
3.9.4.	simple_pair_init	85
3.9.5.	simple_pair_deinit	85
3.9.6.	simple_pair_state_reset	85
3.9.7.	simple_pair_ap_enter_announce_mode	85
3.9.8.	simple_pair_sta_enter_scan_mode	86
3.9.9.	simple_pair_sta_start_negotiate	86
3.9.10.	simple_pair_ap_start_negotiate	86
3.9.11.	simple_pair_ap_refuse_negotiate	86
3.9.12.	simple_pair_set_peer_ref	87
3.9.13.	simple_pair_get_peer_ref	87
3.10. Upg	rade (FOTA) APIs	88

3.10.1. syste	m_upgrade_userbin_check	88
3.10.2. syste	m_upgrade_flag_set	88
3.10.3. syste	m_upgrade_flag_check	88
3.10.4. syste	m_upgrade_start	89
3.10.5. syste	m_upgrade_reboot	89
3.11. Sniffer Rela	ated APIs	90
3.11.1. wifi_p	promiscuous_enable	90
3.11.2. wifi_p	promiscuous_set_mac	90
3.11.3. wifi_s	set_promiscuous_rx_cb	90
3.11.4. wifi_g	get_channel	91
3.11.5. wifi_s	set_channel	91
3.12. Smart Con	fig APIs	92
3.12.1. smart	tconfig_start	92
3.12.2. smart	tconfig_stop	94
3.12.3. smart	tconfig_set_type	94
3.12.4. airkis	s_version	94
3.12.5. airkis	s_lan_recv	95
3.12.6. airkis	s_lan_pack	95
3.13. SNTP APIs	3	96
3.13.1. sntp_	setserver	96
3.13.2. sntp_	getservergetserver	96
3.13.3. sntp_	setservername	96
3.13.4. sntp_	getservername	96
3.13.5. sntp_	init	97
3.13.6. sntp_	stop	97
3.13.7. sntp_	get_current_timestamp	97
3.13.8. sntp_	get_real_time	97
3.13.9. sntp_	set_timezone	97
3.13.10.sntp_	_get_timezone	98
3.13.11.SNTF	P Example	98
3.14. WPA2_Ente	erprise APIs	100
3.14.1. wifi_s	station_set_wpa2_enterprise_auth	100
3.14.2. wifi_s	station_set_enterprise_cert_key	100
3.14.3. wifi_s	station_clear_enterprise_cert_key	101
3.14.4. wifi_s	station_set_enterprise_ca_cert	101
3.14.5. wifi_s	station_clear_enterprise_ca_cert	102

	3.14.6.	wifi_station_set_enterprise_username	102
	3.14.7.	wifi_station_clear_enterprise_username	102
	3.14.8.	wifi_station_set_enterprise_password	102
	3.14.9.	wifi_station_clear_enterprise_password	103
	3.14.10).wifi_station_set_enterprise_new_password	103
	3.14.11	.wifi_station_clear_enterprise_new_password	103
	3.14.12	2.wifi_station_set_enterprise_disable_time_check	103
	3.14.13	B.wifi_station_get_enterprise_disable_time_check	104
	3.14.14	.wpa2_enterprise_set_user_get_time	104
	3.14.15	S.WPA2_Enterprise Work Flow	104
4.	TCP/UDP	APIs	106
	4.1. Gene	eric TCP/UDP APIs	106
	4.1.1.	espconn_delete	106
	4.1.2.	espconn_gethostbyname	106
	4.1.3.	espconn_port	107
	4.1.4.	espconn_regist_sentcb	107
	4.1.5.	espconn_regist_recvcb	108
	4.1.6.	espconn_sent_callback	108
	4.1.7.	espconn_recv_callback	108
	4.1.8.	espconn_get_connection_info	109
	4.1.9.	espconn_send	110
	4.1.10.	espconn_sent	111
	4.2. TCP	APIs	111
	4.2.1.	espconn_accept	111
	4.2.2.	espconn_regist_time	112
	4.2.3.	espconn_connect	112
	4.2.4.	espconn_regist_connectcb	113
	4.2.5.	espconn_connect_callback	113
	4.2.6.	espconn_set_opt	113
	4.2.7.	espconn_clear_opt	114
	4.2.8.	espconn_set_keepalive	114
	4.2.9.	espconn_get_keepalive	115
	4.2.10.	espconn_reconnect_callback	116
	4.2.11.	espconn_regist_reconcb	117
	4.2.12.	espconn_disconnect	117
	4.2.13.	espconn_regist_disconcb	117

4.2.14.	espconn_abort	118
4.2.15.	espconn_regist_write_finish	118
4.2.16.	espconn_tcp_get_max_con	119
4.2.17.	espconn_tcp_set_max_con	119
4.2.18.	espconn_tcp_get_max_con_allow	119
4.2.19.	espconn_tcp_set_max_con_allow	119
4.2.20.	espconn_recv_hold	120
4.2.21.	espconn_recv_unhold	120
4.2.22.	espconn_secure_accept	120
4.2.23.	espconn_secure_delete	121
4.2.24.	espconn_secure_set_size	121
4.2.25.	espconn_secure_get_size	122
4.2.26.	espconn_secure_connect	122
4.2.27.	espconn_secure_send	123
4.2.28.	espconn_secure_sent	123
4.2.29.	espconn_secure_disconnect	124
4.2.30.	espconn_secure_ca_enable	124
4.2.31.	espconn_secure_ca_disable	124
	espconn_secure_cert_req_enable	
4.2.33.	espconn_secure_cert_req_disable	125
4.2.34.	espconn_secure_set_default_certificate	126
4.2.35.	espconn_secure_set_default_private_key	126
4.3. UDP	APIs	126
4.3.1.	espconn_create	126
4.3.2.	espconn_sendto	127
4.3.3.	espconn_igmp_join	127
4.3.4.	espconn_igmp_leave	127
4.3.5.	espconn_dns_setserver	128
4.3.6.	espconn_dns_getserver	128
4.4. mDN	IS APIs	129
4.4.1.	espconn_mdns_init	129
4.4.2.	espconn_mdns_close	129
4.4.3.	espconn_mdns_server_register	129
4.4.4.	espconn_mdns_server_unregister	130
4.4.5.	espconn_mdns_get_servername	130
4.4.6.	espconn_mdns_set_servername	130
4.4.7.	espconn_mdns_set_hostname	130

	4.4.8.	espconn_mdns_get_hostname	130
	4.4.9.	espconn_mdns_enable	131
	4.4.10.	espconn_mdns_disable	131
	4.4.11.	Example of mDNS	131
5.	Application	on-Related APIs	132
	5.1. AT A	Pls	132
	5.1.1.	at_response_ok	132
	5.1.2.	at_response_error	132
	5.1.3.	at_cmd_array_regist	132
	5.1.4.	at_get_next_int_dec	132
	5.1.5.	at_data_str_copy	133
	5.1.6.	at_init	133
	5.1.7.	at_port_print	134
	5.1.8.	at_set_custom_info	134
	5.1.9.	at_enter_special_state	134
	5.1.10.	at_leave_special_state	134
	5.1.11.	at_get_version	135
	5.1.12.	at_register_uart_rx_intr	135
	5.1.13.	at_response	135
	5.1.14.	at_register_response_func	136
	5.1.15.	at_fake_uart_enable	136
	5.1.16.	at_fake_uart_rx	136
	5.1.17.	at_set_escape_character	136
	5.2. Relat	ted JSON APIs	137
	5.2.1.	jsonparse_setup	137
	5.2.2.	jsonparse_next	137
	5.2.3.	jsonparse_copy_value	137
	5.2.4.	jsonparse_get_value_as_int	138
	5.2.5.	jsonparse_get_value_as_long	138
	5.2.6.	jsonparse_get_len	138
	5.2.7.	jsonparse_get_value_as_type	138
	5.2.8.	jsonparse_strcmp_value	138
	5.2.9.	jsontree_set_up	139
	5.2.10.	jsontree_reset	139
	5.2.11.	jsontree_path_name	139
	5.2.12.	jsontree_write_int	139

	5.2.13.	jsontree_write_int_array	140
	5.2.14.	jsontree_write_string	140
	5.2.15.	jsontree_print_next	140
	5.2.16.	jsontree_find_next	141
6.	Definition	ns & Structures	142
	6.1. Time	er	142
	6.2. Wi-F	Fi-Related Structures	142
	6.2.1.	Station Parameters	142
	6.2.2.	SoftAP Parameters	142
	6.2.3.	Scan Parameters	143
	6.2.4.	Wi-Fi Event-Related Structures	144
	6.2.5.	SmartConfig Structures	146
	6.3. JSO	N-Related Structure	146
	6.3.1.	JSON Structures	146
	6.4. espo	conn Parameters	148
	6.4.1.	Callback Functions	148
	6.4.2.	espconn Structures	148
	6.4.3.	Interrupt-Related Definitions	149
7.	Periphera	al-Related Drivers	151
	7.1. GPI	O Related APIs	151
	7.1.1.	PIN-Related Macros	151
	7.1.2.	gpio_output_set	151
	7.1.3.	GPIO input and output macros	152
	7.1.4.	GPIO interrupt	152
	7.1.5.	gpio_pin_intr_state_set	152
	7.1.6.	GPIO Interrupt Handler	152
	7.2. UAF	RT-Related APIs	153
	7.2.1.	uart_init	153
	7.2.2.	uart0_tx_buffer	153
	7.2.3.	uart0_rx_intr_handler	153
	7.2.4.	uart_div_modify	154
	7.3. I2C	Master-Related APIs	154
	7.3.1.	i2c_master_gpio_init	154
	7.3.2.	i2c_master_init	154
	7.3.3.	i2c_master_start	154

	7.3.4.	i2c_master_stop	155
	7.3.5.	i2c_master_send_ack	155
	7.3.6.	i2c_master_send_nack	155
	7.3.7.	i2c_master_checkAck	155
	7.3.8.	i2c_master_readByte	155
	7.3.9.	i2c_master_writeByte	156
	7.4. PWN	1/-Related APIs	156
	7.4.1.	pwm_init	156
	7.4.2.	pwm_start	157
	7.4.3.	pwm_set_duty	157
	7.4.4.	pwm_get_duty	157
	7.4.5.	pwm_set_period	157
	7.4.6.	pwm_get_period	158
	7.4.7.	get_pwm_version	158
	7.5. SDIC) APIs	158
	7.5.1.	sdio_slave_init	158
	7.5.2.	sdio_load_data	158
	7.5.3.	sdio_register_recv_cb	159
A.	Appendix		160
	A.1. ESP	CONN Programming	160
	A.1.1.	TCP Client Mode	160
	A.1.2.	TCP Server Mode	160
	A.1.3.	espconn callback	161
	A.2. RTC	APIs Example	161
	A.3. Sniff	er Introduction	163
	A.4. ESP	8266 SoftAP and Station Channel Configuration	163
		8266 Boot Messages	
		8266 Signaling Measurement	



1.

Preambles

ESP8266 Wi-Fi SoC offers a complete and self-contained Wi-Fi networking solution; it can be used to host applications or to offload Wi-Fi networking functions from another application processor. When the ESP8266 hosts application, it boots up directly from an external flash. It has an integrated cache to improve the performance of system's running applications. Alternately, serving as a Wi-Fi adapter, wireless internet access can be added into any microcontroller-based design with simple connectivity through UART interface or the CPU AHB bridge interface.

ESP8266EX is amongst the most integrated Wi-Fi chips in the industry; it integrates the antenna switches, RF balun, power amplifier, low noise receive amplifier, filters, and power management modules. Thus, it requires minimal external circuitry, and the entire solution, including front-end module, is designed to occupy minimal PCB area.

ESP8266EX also integrates an enhanced version of Tensilica's L106 Diamond series 32-bit processor, with on-chip SRAM, on top of its Wi-Fi functionalities. It is integrated with external sensors and other application specific devices through its GPIOs. Codes for such applications are provided as examples in the SDK.

Sophisticated system-level features include fast sleep/wake switching for energy-efficient VoIP, adaptive radio biasing for low-power operations, advanced signal processing, spur cancellation and radio co-existence features for common cellular, Bluetooth, DDR, LVDS, LCD interference mitigation.

The SDK based on ESP8266 IoT platform offers the users an easy, fast and efficient way to develop IoT devices. This programming guide provides overview of the SDK as well as detailed description of the APIs. It is written to help embedded software developers program on ESP8266 IoT platform.



2.

Non-OS SDK

2.1. Non-OS SDK Introduction

The non-OS SDK provides a set of application programming interfaces (APIs) for core ESP8266 functionalities such as data reception/transmission over Wi-Fi, TCP/IP stack functions, hardware interface functions and basic system management functions. Thus, the SDK APIs allow programmers to focus on the application under development at a higher level. Users can access all core capabilities of ESP8266 without studying its chip architecture in detail.

All networking functions on the ESP8266 IoT platform are realized in the library, and are available to users in the form of well-defined APIs. Users can initialize the system hardware and network interface in *user_main.c* when ESP8266 boots up.

void user_init (void) is the default method provided. Users can add functions like
firmware initialization, network parameters setting, and timer initialization within user_init.

void user_rf_pre_init (void) and uint32 user_rf_cal_sector_set(void) need to be added to user_main.c from ESP8266_NONOS_SDK_V1.5.2 onwards. For an example of how to set the RF calibration sector, please refer to ESP8266_NONOS_SDK/examples/IOT_Demo/user/user_main.c. Users can call system_phy_set_rfoption to set the RF option in user_rf_pre_init, or call system_deep_sleep_set_option before Deep-sleep. If RF is disabled, ESP8266 Station and SoftAP will both be disabled, so the related APIs must not be called. Wi-Fi radio functions and network stack management APIs are not available when the radio is disabled.

From *ESP8266_NONOS_SDK_V2.1.0* onwards, when the DIO-to-QIO flash is not used, users can add an empty function void user_spi_flash_dio_to_qio_pre_init(void) on the application side to reduce iRAM usage.

APIs for JSON packet construction and parsing are included in the SDK. Users themselves can define the format of data packets.

2.2. Code Structure

The non-OS SDK is meant to be used for applications where users require complete control over the execution sequence of code. Due to a lack of operating system, the non-OS SDK does not schedule tasks or preempt the user functions.

The non-OS SDK is most suitable for use in event-driven applications. As there is no RTOS overhead, the non-OS SDK does not impose stack size restrictions or execution time slots on any user functions.

The RTOS SDK is an alternative to non-OS SDK, and may be used where task-based, modular programming is desirable. To read more about the RTOS SDK, please refer to the *ESP8266 SDK Getting Started Guide*.

The code structure in non-OS SDK may be described as follows:



- The non-OS SDK does not implement user task scheduling like RTOS based systems do. The non-OS SDK uses four types of functions:
 - Application functions
 - Callback functions
 - Interrupt service routines (ISRs)
 - User tasks

Application functions refer to the usual type of C functions used in embedded C programming. These functions must be called by another function. Application functions may be attributed with <code>ICACHE_FLASH_ATTR</code> to fetch and execute programs from the flash. <code>IRAM_ATTR</code>-attributed functions are stored in the iRAM prior to execution.

Callback functions refer to functions that are not called directly from the user program. Callback functions are executed by the non-OS SDK core when a system event occurs. This enables the programmer to respond to real-time events without using RTOS or polling for events.

To program a callback function, users first need to register the callback function using the corresponding register_cb API. Examples of callback functions include timer callback functions and network event callback functions.

Interrupt Service Routines (ISRs) are simply callback functions of a special type. These functions are called when a hardware interrupt occurs. When an interrupt is enabled, a corresponding interrupt handler functions must be registered. Note that ISRs must be attributed as IRAM ATTR.

User tasks can be classified according to three priority levels: 0, 1, 2. Priority level has the following order: 2>1>0. Non-OS SDK can only support up to three tasks at a time. One priority level for one task.

User tasks are normally used when a function cannot be called directly. To create a user task, please refer to the API description of <code>system_os_task()</code> in this document. For example, <code>espconn_disconnect()</code> API may not be called from within an <code>espconn</code> callback, therefore a user task must be created within the <code>espconn</code> callback to <code>execute espconn_disconnect</code>.

- As stated earlier, the non-OS SDK does not preempt tasks or switch context. Users
 are responsible for the proper execution of code and the user code must not occupy
 the CPU on a particular function for too long. This may cause a watchdog reset and
 prompt ESP8266 to reboot.
 - If for some reason the user application must execute a task for too long (say, longer than 500 ms), it is recommended that the <code>system_soft_wdt_feed()</code> API be called often to reset the WDT. Disabling the softWDT is not recommended.
- For proper system operation, it is highly recommended that the *esp_init_data.bin* and *blank.bin* files be flashed to the correct location in memory at least once. The RF calibration sector must be set correctly in the application user_rf_cal_sector_set.



2.3. Timer and Interrupt

- For polling purposes, it is highly recommended that system timer task be used check for an event periodically.
 - Using a loop is inefficient and blocks the CPU.
 - Users need to call function os_delay_us or while, or for in timer callback. Please do not occupy CPU for more than 15 ms.
- Timer APIs may not be used to generate a callback more frequently than every 5 ms (100 µs for microsecond timer). Please refer to descriptions of os_timer_arm() and related APIs for detailed information on timer usage.
- The microsecond timer is not very precise. Even though timer callbacks are assigned a higher priority over user functions, there might be a 500 µs jitter in callbacks. For high accuracy timing purposes, please use the hardware timer by referring to the driver program (*driver_lib*). Note that PWM APIs may not be used when the hardware timer is in use.
- Do not disable interrupts for too long. ISRs should occupy very little time as well (in microseconds only).

2.4. System Performance

- ESP8266 is typically run at 80 MHz. The CPU may be configured to run at 160 MHz in high performance applications. Note that the peripherals are not affected by the CPU frequency setting because they are run off a different clock source from that of the CPU.
- Higher clock frequency and disabled sleep modes cause greater power consumption in return for better performance. The user application should prioritize between the two.
- Code attributed with ICACHE_FLASH_ATTR is generally executed slower than the code marked with IRAM_ATTR. However, like most embedded platforms, iRAM is limited on ESP8266 and thus must be used sparingly for high performance code.
- The flash mode and frequency directly influence the code execution speed. Setting the flash to a higher frequency and QIO mode may produce the best results in terms of performance, though it costs in terms of power consumption.

2.5. System Memory

- ESP8266 supports a primary standard external QSPI flash memory of up to 128 Mbits for code and data storage. There may be secondary memory chips for user data storage as well.
- There is no internal non-volatile storage within ESP8266 for storing user code or data.
 The ESP8285 chip is an ESP8266 variant with integrated flash memory. Please refer to the ESP8285 Datasheet for more details.
- ESP8266 features 160 Kbytes of RAM, of which iRAM is 64 KB and dRAM is 96KB (dRAM: 80 KB for SDK+HEAP, 16 KB for ROM). The iRAM is further segmented into two

Espressif 4/166 2018.08



- blocks. A 32 KB iRAM block stores code marked with IRAM_ATTR and the other 32 KB block is used to cache code from flash, i.e. code marked with ICACHE_FLASH_ATTR.
- Since ESP8266_NonOS_SDK_V3.0, we have added a new feature to enable using iRAM as memory, which can provide about 17KB extra memory. Please find the details below. Please note that it may effect the system performance, so it's recommended to conduct a thorough test to ensure what works best for you.
 - Define function user_iram_memory_is_enabled in application and return 1 to enable iRAM as memory. For example,

```
#define CONFIG_ENABLE_IRAM_MEMORY

#ifdef CONFIG_ENABLE_IRAM_MEMORY

uint32 user_iram_memory_is_enabled(void)

{
    return CONFIG_ENABLE_IRAM_MEMORY;
}
#endif
```

- After setting as above, iRAM is enabled as the first chosen memory by default. os_malloc, os_zalloc and os_calloc will allocate from iRAM first, and dRAM will be the next available memory when iRAM is used up.
- Or users can directly call os_malloc_iram, os_zalloc_iram, os_calloc_iram to allocate from iRAM (dRAM will be the next available memory when iRAM is used up); and os_malloc_dram, os_zalloc_dram, os_calloc_dram to allocate from dRAM.
- Or to be compatible with earlier applications, users can define MEM_DEFAULT_USE_DRAM to make os_malloc, os_zalloc and os_calloc allocate from dRAM, and use os_malloc_iram, os_zalloc_iram, os_calloc_iram to allocate from iRAM. For example, add below codes in makefile:

```
CONFIGURATION_DEFINES += -DMEM_DEFAULT_USE_DRAM
```

The definition in include/mem.h is as below:

```
#ifdef MEM_DEFAULT_USE_DRAM

#define os_malloc os_malloc_dram

#define os_zalloc os_zalloc_dram

#define os_calloc os_calloc_dram

#else

#define os_malloc os_malloc_iram

#define os_zalloc os_zalloc_iram

#define os_zalloc os_zalloc_iram

#define os_calloc os_calloc_iram
```



• RAM and flash access have to be word-aligned (4 byte boundary aligned access only). Casting pointers directly is not recommended. Please use os_memcpy, or other APIs for memory operations.



3. Application Programming Interface (APIs)

3.1. Software Timer

Timer APIs can be found in /ESP8266_NONOS_SDK/include/osapi.h.

Please note that os_timer APIs listed below are software timers executed in tasks, thus timer callbacks may not be precisely executed at the right time; it depends on priority. If you need a precise timer, please use a hardware timer which can be executed in hardware interrupt. For details please refer to **hw_timer.c**.

- For the same timer, os_timer_arm (or os_timer_arm_us) cannot be invoked repeatedly. os_timer_disarm should be invoked first.
- os_timer_setfn can only be invoked when the timer is not enabled, i.e., after os_timer_disarm or before os_timer_arm (or os_timer_arm_us).

3.1.1. os_timer_arm

Function	Enable a millisecond timer.
Prototype	<pre>void os_timer_arm (os_timer_t *ptimer, uint32_t milliseconds, bool repeat_flag)</pre>
Parameter	os_timer_t *ptimer: timer structure. uint32_t milliseconds: timing; unit: millisecond. If system_timer_reinit has been called, the timer value allowed ranges from 100 to 0x689D0. If system_timer_reinit has NOT been called, the timer value allowed ranges from 5 to 0x68D7A3. bool repeat_flag: whether the timer will be invoked repeatedly or not.
Return	none

3.1.2. os timer disarm

Function	Disarm the timer.
Prototype	void os_timer_disarm (os_timer_t *ptimer)
Parameter	os_timer_t *ptimer: timer structure.
Return	none

Espressif 7/158 2018.08



3.1.3. os_timer_setfn

Function	Set timer callback function. The timer callback function must be set before arming a timer.
Prototype	<pre>void os_timer_setfn(os_timer_t *ptimer, os_timer_func_t *pfunction, void *parg)</pre>
Parameter	os_timer_t *ptimer: timer structure. os_timer_func_t *pfunction: timer callback function; use typecasting to pass function as your function. void *parg: callback function parameter.
Return	none

3.1.4. system_timer_reinit

Function	Reinitiate the timer when you need to use microsecond timer.
Prototype	void system_timer_reinit (void)
Parameter	none
Return	none
Note	1. Define USE_US_TIMER;
11010	2. Put system_timer_reinit at the beginning of user_init, in the first sentence.

3.1.5. os_timer_arm_us

Function	Enable a microsecond timer.
Prototype	<pre>void os_timer_arm_us (os_timer_t *ptimer, uint32_t microseconds, bool repeat_flag)</pre>
Parameter	os_timer_t *ptimer: timer structure. uint32_t microseconds: timing; unit: microsecond, the minimum value is 0x64, the maximum value allowed to input is 0xFFFFFF. bool repeat_flag: whether the timer will be invoked repeatedly or not.
Return	none
Note	 Define USE_US_TIMER, and put system_timer_reinit at the beginning of user_init, in the first sentence. The highest precision is 500 µs.



3.2. Hardware Timer

Hardware timer APIs can be found in /ESP8266_NONOS_SDK/examples/driver_lib/hw_timer.c. Users can use it according to driver_lib/readme.txt.

Notes:

- If NM is used as the ISR source for auto-loading the timer, parameter val of hw_timer_arm can not be less than 100.
- When NMI source is used, the timer has the highest priority. It can interrupt other ISRs. FRC1 source should be used to prevent the timer from interrupting other ISRs.
- APIs in hw_timer.c can not be called when PWM APIs are in use, because they all use the same hardware timer.
- The hardware timer callback function must NOT be defined with ICACHE_FLASH_ATTR.
- The system must not be allowed to enter Light-sleep mode (wifi_set_sleep_type(LIGHT_SLEEP)) when hardware timer is enabled. Light-sleep stops the CPU and it can not be interrupted by NMI.

3.2.1. hw_timer_init

Function	Initialize the hardware ISR timer.
Prototype	<pre>void hw_timer_init (FRC1_TIMER_SOURCE_TYPE source_type, u8 req)</pre>
	FRC1_TIMER_SOURCE_TYPE source_type: ISR source of timer.
Parameter	FRC1_SOURCE: timer uses FRC1 ISR as ISR source.
i didiliotoi	NMI_SOURCE: timer uses NMI ISR as ISR source.
	u8 req: 0: autoload disabled; 1: autoload enabled.
Return	none

3.2.2. hw_timer_arm

Function	Set a trigger timer delay to enable this timer.
Prototype	void hw_timer_arm (uint32 val)
	uint32 val: timing
	In autoload mode:
Parameter	- For FRC1_SOURCE, range: 50 ~ 0x199999 μs;
	- For NMI_SOURCE, range: 100 ~ 0x199999 μs.
	 In non autoload mode, range: 10 ~ 0x199999 μs.
Return	none

Espressif 9/158 2018.08



3.2.3. hw_timer_set_func

Function	Set timer callback function.
runction	The timer callback function must be set before arming a timer.
Prototype	<pre>void hw_timer_set_func (void (* user_hw_timer_cb_set)(void))</pre>
Parameters	<pre>void (* user_hw_timer_cb_set)(void): timer callback function, must NOT be defined as ICACHE_FLASH_ATTR.</pre>
Return	none
Notes	ICACHE_FLASH_ATTR is not allowed to be added before the timer callback.

3.2.4. Hardware Timer Example

```
#define REG_READ(_r)
                         (*(volatile uint32 *)(_r))
#define WDEV_NOW()
                      REG_READ(0x3ff20c00)
uint32 tick_now2 = 0;
void hw_test_timer_cb(void)
    static uint16 j = 0;
    j++;
    if( (WDEV_NOW() - tick_now2) >= 1000000 )
        static u32 idx = 1;
        tick_now2 = WDEV_NOW();
        os_printf("b%u:%d\n",idx++,j);
        j = 0;
    }
}
void ICACHE_FLASH_ATTR user_init(void)
{
        hw_timer_init(FRC1_SOURCE,1);
        hw_timer_set_func(hw_test_timer_cb);
        hw_timer_arm(100);
}
```



3.3. System APIs

System APIs can be found in /ESP8266_NONOS_SDK/include/user_interface.h. os_XXX APIs can be found in /ESP8266_NONOS_SDK/include/osapi.h.

3.3.1. system_get_sdk_version

Function	Get SDK version.
Prototype	const char* system_get_sdk_version(void)
Parameter	none
Return	SDK version
Example	os_printf("SDK version: %s \n", system_get_sdk_version());

3.3.2. system_restore

Function	Reset default settings of following APIs: wifi_station_set_auto_connect, wifi_set_phy_mode, wifi_softap_set_config related, wifi_station_set_config related, wifi_set_opmode, and APs' information recorded by #define AP_CACHE.
Prototype	void system_restore(void)
Parameter	none
Return	none
Note	Call system_restart to restart after reset by system_restore.

3.3.3. system_restart

Function	Restart.
Prototype	void system_restart(void)
Parameter	none
Return	none
Note	The ESP8266 will not restart immediately. Please do not call other functions after calling this API.

3.3.4. system_init_done_cb

Function	Call this API in user_init to register a system-init-done callback.
Prototype	<pre>void system_init_done_cb(init_done_cb_t cb)</pre>
Parameter	init_done_cb_t cb: system_init_done callback.
Return	none

Espressif 11/158 2018.08



```
void to_scan(void) { wifi_station_scan(NULL,scan_done); }
void user_init(void) {
    wifi_set_opmode(STATION_MODE);
    system_init_done_cb(to_scan);
}
```

3.3.5. system_get_chip_id

Function	Get chip ID.
Prototype	uint32 system_get_chip_id (void)
Parameter	none
Return	Chip ID.

3.3.6. system_get_vdd33

Function	Measure the power voltage of VDD3P3 pin 3 and 4; unit: 1/1024 V.
Prototype	uint16 system_get_vdd33(void)
Parameter	none
Return	Power voltage of VDD33; unit: 1/1024 V.
	system_get_vdd33 can only be called when TOUT pin is suspended.
Note	 The 107th byte in esp_init_data_default.bin (0 ~ 127 bytes) is named as vdd33_const. When TOUT pin is suspended, vdd33_const must be set as 0xFF, which is 255.
	The return value of system_get_vdd33 may be different in different Wi-Fi modes, for example, in Modem-sleep mode or in normal Wi-Fi working mode.

3.3.7. system_adc_read

Function	Measure the input voltage of TOUT pin 6; unit: 1/1024 V.
Prototype	uint16 system_adc_read(void)
Parameter	none
Return	Input voltage of TOUT pin 6; unit:1/1024 V.

Espressif 12/158 2018.08



	• system_adc_read is only available when TOUT pin is wired to external circuitry. Input Voltage Range restricted to 0 ~ 1.0V.
	 The 107th byte in esp_init_data_default.bin (0 ~ 127 bytes) is named as vdd33_const, and when TOUT pin is wired to external circuitry, the vdd33_const must be set as real power voltage of VDD3P3 pin 3 and 4, and has to be less than 0xFF.
Note	 The range of operating voltage of ESP8266 is 1.8V ~ 3.6V, the unit of vdd33_const is 0.1V, so effective value range of vdd33_const is [18, 36]. If vdd33_const is an ineffective value in the range of (0, 18) or (36, 255), ESP8266 RF calibration will be 3.3V by default.
	The return value of system_adc_read may be different in different Wi-Fi modes, for example, in Modem-sleep mode or in normal Wi-Fi working mode.
	If high precision is needed, please use system_adc_read_fast instead.

3.3.8. system_adc_read_fast

Function	Fast and high-precision sampling of ADC.
Prototype	<pre>void system_adc_read_fast (uint16 *adc_addr, uint16 adc_num, uint8 adc_clk_div)</pre>
	uint16 *adc_addr: point to the address of ADC continuously fast sampling output.
Parameter	uint16 adc_num: sampling number of ADC continuously fast sampling; range [1, 65535].
	uint8 adc_clk_div: ADC working clock = 80M/adc_clk_div; range [8, 32], the recommended value is 8.
Return	none

Espressif 13/158 2018.08



```
extern void system_adc_read_fast(uint16 *adc_addr, uint16 adc_num,
                  uint8 adc_clk_div);
                  os_timer_t timer;
                  void ICACHE_FLASH_ATTR ADC_TEST(void *p)
                          wifi_set_opmode(NULL_MODE);
                          ets_intr_lock( );
                                                    //close interrupt
                          uint16 adc_addr[10];
                          uint16 adc_num = 10;
                          uint8 adc_clk_div = 8;
Example
                          uint32 i;
                           system_adc_read_fast(adc_addr, adc_num, adc_clk_div);
                           for(i=0; i<adc_num; i++)</pre>
                                   os_printf("i=%d, adc_v=%d\n", i, adc_addr[i]);
                           ets_intr_unlock();
                                                             //open interrupt
                          os_timer_disarm(&timer);
                          os_timer_setfn(&timer, ADC_TEST, NULL);
                          os_timer_arm(&timer,1000,1);
                  }
                  • To use system_adc_read_fast, Wi-Fi has to be disabled. And if ADC
                    continuously sampling is needed, all interrupts have to be disabled, so PWM or
                     NMI hardware timer can not be used when system_adc_read_fast is calling.
                    system_adc_read_fast is only available when TOUT pin is wired to external
                     circuitry. Input voltage range is restricted to 0 ~ 1.0V.
                  • The [107] byte in esp_init_data_default.bin (0 ~ 127 bytes) is named as
                     vdd33_const, and when TOUT pin is wired to external circuitry working as an
                     ADC input, the vdd33_const must be less than 0xFF.
                  • Details of the [107] byte vdd33_const are as follows:
  Note
                     - If the [107] byte = 0XFF, VDD33 is measured internally, TOUT pin cannot
                        work as ADC input.
                     - When the valid value range of the [107] byte vdd33_const is [18, 36], the
                        unit is 0.1V, and the value should be set as the real power voltage of
                        VDD33 to optimize the RF working condition. TOUT pin can work as ADC
                       When the [107] byte vdd33_const is in the range of (0, 18) or (36, 255), the
                        default 3.3V will be used as the power voltage to optimize the RF working
                        condition. TOUT pin can work as ADC input.
```

3.3.9. system_deep_sleep

Function	Configures chip for Deep-sleep mode. When the device is in Deep-sleep, it automatically wakes up when time out. Upon waking up, the device boots up from user_init.	
Prototype	bool system_deep_sleep(uint64 time_in_us)	



	uint64 time_in_us: the duration of time (µs) when the device is in Deep-sleep.
	The theoretical maximum value of time_in_us can be calculated by formula: (time_in_us / cali) << 12 = 2^31 - 1
Parameter	• cali = system_rtc_clock_cali_proc(), the cali is the RTC clock period (in us); bit11 ~ bit0 are decimal. For more details about the cali, please see the API: system_rtc_clock_cali_proc.
	The input value of time_in_us should be less than the theoretical maximum value.
Return	true: Success
rictarri	false: Failure
	• Hardware has to support deep-sleep wake up (XPD_DCDC connects to EXT_RSTB with a $0-\Omega$ resistor).
Note	• system_deep_sleep(0): there is no wakeup timer; in order to wake up, connect a GPIO to pin RST; the chip will wake up by a falling-edge on pin RST.
	After configuration, the chip will not enter Deep-sleep mode immediately, but will wait for a while till the Wi-Fi core is closed safely.

${\it 3.3.10. system_deep_sleep_set_option}$

Function	Call this API before system_deep_sleep to set whether the chip will do RF calibration or not when it wakes up from deep-sleep again. The option is 1 by default.
Prototype	bool system_deep_sleep_set_option(uint8 option)
	uint8 option:
	0: RF calibration after deep-sleep wakeup depends on both the times of entering Deep-sleep (deep_sleep_number, returns 0 upon each power-up) and byte 108 of esp_init_data_default.bin (0 ~ 127 bytes).
	• If deep_sleep_number <= byte 108, no RF calibration after Deep-sleep wakeup; this reduces the current consumption.
Parameter	• If deep_sleep_number = byte 108 +1, the behavior after Deep-sleep wakeup will be the same as power-up, and deep_sleep_number returns to 0.
	- The behavior after Deep-sleep wakeup will be the same as power-up.
	 No RF calibration after Deep-sleep wakeup; this reduces the current consumption.
	 Disable RF after Deep-sleep wakeup, just like Modem-sleep; this has the least current consumption; the device is not able to transmit or receive data after wakeup.
Return	true: Success
neturii	false: Failure
Note	 Hardware has to support deep-sleep wake up (XPD_DCDC connects to EXT_RSTB with a 0-Ω resistor).
Hote	• system_deep_sleep(0): there is no wakeup timer; in order to wake up, connect a GPIO to pin RST; the chip will wake up by a falling-edge on pin RST.

Espressif 15/158 2018.08



3.3.11. system_phy_set_rfoption

Function	Enable RF or not when wakeup from Deep-sleep.
Prototype	void system_phy_set_rfoption(uint8 option)
	 uint8 option: 0: RF calibration after deep-sleep wakeup depends on both the times of entering Deep-sleep (deep_sleep_number, returns 0 upon each power-up) and byte 108 of esp_init_data_default.bin (0 ~ 127 bytes).
	- If deep_sleep_number <= byte 108, no RF calibration after wakeup from Deep-sleep; this reduces the current consumption.
Parameter	 If deep_sleep_number = byte 108 +1, the behavior after Deep-sleep wakeup will be the same as power-up, and deep_sleep_number becomes 0.
	The behavior after Deep-sleep wakeup will be the same as power-up.
	No RF calibration after Deep-sleep wakeup; this reduces the current consumption.
	Disable RF after Deep-sleep wakeup, just like modem sleep; this has the least current consumption; the device is not able to transmit or receive data after wakeup.
Return	none
	This API can only be called in user_rf_pre_init.
Note	• Function of this API is similar to system_deep_sleep_set_option. If called, it will disregard system_deep_sleep_set_option which is called before Deep-sleep, and refer to system_phy_set_rfoption which is called upon Deep-sleep wakeup.
	Before calling this API, system_deep_sleep_set_option should be called at least once.

3.3.12. system_phy_set_powerup_option

Function	Set whether the chip will do RF calibration or not when power up. The option is 0 by default.
Prototype	void system_phy_set_powerup_option(uint8 option)
Parameter	 uint8 option: RF initialization upon powerup. 0: RF initialization when powerup depends on byte 114 of esp_init_data_default.bin (0 ~ 127 bytes). For more details please see ESP8266 SDK Getting Started Guide. 1: RF initialization only calibrate VDD33 and Tx power which will take about 18 ms; this reduces the current consumption.
	 2: RF initialization only calibrate VDD33 which will take about 2 ms; this has the least current consumption. 3: RF initialization will do the whole RF calibration which will take about 200 ms; this increases the current consumption.
Return	none

Espressif 16/158 2018.08



3.3.13. system_phy_set_max_tpw

Function	Set maximum value of RF Tx Power; unit : 0.25 dBm.
Prototype	void system_phy_set_max_tpw(uint8 max_tpw)
Parameter	uint8 max_tpw: maximum value of RF Tx Power, unit: 0.25 dBm, range [0, 82]. It can be set by referring to the 34th byte (target_power_qdb_0) of esp_init_data_default.bin (0 ~ 127 bytes).
Return	none

3.3.14. system_phy_set_tpw_via_vdd33

Function	Adjust RF Tx Power according to VDD33; unit : 1/1024 V.
Prototype	void system_phy_set_tpw_via_vdd33(uint16 vdd33)
Parameter	uint16 vdd33: VDD33, unit : 1/1024V, range [1900, 3300]
Return	none
Note	When TOUT pin is suspended, VDD33 can be got by system_get_vdd33;
	When TOUT pin is wired to external circuitry, system_get_vdd33 can not be used.

3.3.15. system_set_os_print

Function	Turn log printing on or off.
Prototype	void system_set_os_print (uint8 onoff)
Parameter	uint8 onoff
Return	none
Note	onoff = 0: print function off onoff = 1: print function on
Default	Print function on.

3.3.16. system_print_meminfo

Function	Print memory information, including data/rodata/bss/heap.
Prototype	void system_print_meminfo (void)
Parameter	none
Return	none

Espressif 17/158 2018.08



3.3.17. system_get_free_heap_size

Function	Get free heap size.
Prototype	uint32 system_get_free_heap_size(void)
Parameter	none
Return	uint32: available heap size

3.3.18. system_os_task

Function	Set up tasks.
Prototype	bool system_os_task(os_task_t task, uint8 prio, os_event_t *queue, uint8 qlen)
Parameters	os_task_t task: task function. uint8 prio: task priority. Three priorities are supported: 0/1/2; 0 is the lowest priority. This means only 3 tasks are allowed to be set up. os_event_t *queue: message queue pointer. uint8 qlen: message queue depth.
Return	true:Success false:Failure
Example	<pre>#define SIG_RX</pre>



3.3.19. system_os_post

Function	Send messages to task.
Prototype	<pre>bool system_os_post (uint8 prio, os_signal_t sig, os_param_t par)</pre>
Parameter	uint8 prio: task priority, corresponding to that you set up. os_signal_t sig: message type. os_param_t par: message parameters.
Return	true:Success false:Failure
Example	<pre>void task_post(void) { system_os_post(USER_TASK_PRIO_0, SIG_RX, 'a'); }</pre>
Print	sig_rx a

3.3.20. system_get_time

Function	Get system time (µs).
Prototype	uint32 system_get_time(void)
Parameter	none
Return	System time in microsecond.

3.3.21. system_get_rtc_time

Function	Get RTC time, as denoted by the number of RTC clock periods.
Prototype	uint32 system_get_rtc_time(void)
Parameter	none
Return	RTC time.
Example	If $system_get_rtc_time$ returns 10 (which means 10 RTC cycles), and $system_rtc_clock_cali_proc$ returns 5.75 (which means 5.75 μ s per RTC cycle), the real time is $10 \times 5.75 = 57.5 \mu$ s.

Espressif 19/158 2018.08



	System time will return to zero because of system_restart, but RTC still goes on. If an external hardware reset the chip via EXT_RST or CHIP_EN (such as timed wakeup from Deep-sleep), the RTC timer will be reset.
	Reset by pin EXT_RST: RTC memory won't change; RTC timer starts to zero.
Note	Watchdog reset: RTC memory won't change; RTC timer won't change.
	• system_restart: RTC memory won't change; RTC timer won't change.
	Power-on: RTC memory contains a random value; RTC timer starts from zero.
	 Reset by pin CHIP_EN: RTC memory contains a random value; RTC timer starts from zero.

3.3.22. system_rtc_clock_cali_proc

Function	Get RTC clock period.
Prototype	uint32 system_get_rtc_time(void)
Parameter	none
Return	RTC clock period (in us); bit11 ~ bit0 are decimal.
Example	os_printf("clk cal : %d \r\n",system_rtc_clock_cali_proc()>>12);
	RTC clock period has decimal part.
Note	RTC clock period tends to drift with changes in temperature, so RTC timer is not very precise.
	See RTC demo in <i>Appendix.A</i> .

$3.3.23.\ system_rtc_mem_write$

	Writes data to the RTC memory.
Function	During Deep-sleep mode, the RTC is still operational and can store user data in the defined user-data area.
	<system (256="" bytes)="" data=""> <user (512="" bytes)="" data=""> </user></system>
Prototype	<pre>bool system_rtc_mem_write (uint32 des_addr, void * src_addr, uint32 save_size)</pre>
Parameter	<pre>uint32 des_addr: destination address (block number) in RTC memory, des_addr >=64 void * src_addr: data pointer uint32 save_size: data length (byte)</pre>
Return	true: Success false: Failure
Example	os_printf("clk cal : %d \r\n",system_rtc_clock_cali_proc()>>12);

Espressif 20/158 2018.08



Note	Data read/write accesses to the RTC memory must be word-aligned (4-byte boundary aligned). Parameter <code>des_addr</code> means block number (4 bytes per block). For example, to save data at the beginning of user data area, <code>des_addr</code> will be 256/4 = 64, and <code>save_size</code> will be data length.

3.3.24. system_rtc_mem_read

Function	Read user data from RTC memory. Only user data area should be accessed by the user. <system (256="" bytes)="" data=""> <user (512="" bytes)="" data=""> </user></system>
Prototype	<pre>bool system_rtc_mem_read (uint32 src_addr, void * des_addr, uint32 save_size)</pre>
Parameter	<pre>uint32 src_addr: source address (block number) in RTC memory, src_addr >= 64 void * des_addr: data pointer uint32 save_size: data length; unit: byte</pre>
Return	true: Success false: Failure
Note	Data read/write accesses to the RTC memory must be word aligned (4 bytes boundary aligned). Parameter <pre>src_addr</pre> means block number(4 bytes per block). For example, to read data from the beginning of user data area, <pre>src_addr</pre> will be 256/4=64, <pre>save_size</pre> will be data length.

3.3.25. system_uart_swap

Function	UARTO swap. Use MTCK as UARTO Rx, MTDO as UARTO Tx, so ROM log will not output from this new UARTO. MTDO (UORTS) and MTCK (UOCTS) also need to be used as UARTO in hardware.
Prototype	void system_uart_swap (void)
Parameter	none
Return	none

3.3.26. system_uart_de_swap

Function	Disable UART0 swap. Use original UART0, not MTCK and MTDO.
Prototype	void system_uart_de_swap (void)
Parameter	none
Return	none

Espressif 21/158 2018.08



3.3.27. system_get_boot_version

Function	Get version info of boot.
Prototype	uint8 system_get_boot_version (void)
Parameter	none
Return	Version info of boot.
Note	If boot version >= 3, it is possible to enable enhanced boot mode (for details of which please see system_restart_enhance).

3.3.28. system_get_userbin_addr

Function	Get address of the current running user bin (user1.bin or user2.bin).
Prototype	uint32 system_get_userbin_addr (void)
Parameter	none
Return	Start address info of the current running user binary.

3.3.29. system_get_boot_mode

Function	Get boot mode.
Prototype	uint8 system_get_boot_mode (void)
Parameter	none
Return	#define SYS_BOOT_ENHANCE_MODE 0 #define SYS_BOOT_NORMAL_MODE 1
Note	 Enhance boot mode: can load and run FW at any address. Normal boot mode: can only load and run normal <i>user1.bin</i> (or <i>user2.bin</i>).

3.3.30. system_restart_enhance

Function	Restarts system, and enters enhance boot mode.
Prototype	<pre>bool system_restart_enhance(uint8 bin_type, uint32 bin_addr)</pre>
Parameter	<pre>uint8 bin_type: type of bin #define SYS_BOOT_NORMAL_BIN 0 // user1.bin or user2.bin #define SYS_BOOT_TEST_BIN 1 // can only be Espressif test bin uint32 bin_addr: start address of bin file</pre>

Espressif 22/158 2018.08



Return	true: Success false: Failure
Note	SYS_BOOT_TEST_BIN is for factory test during production; you can apply for the test bin from Espressif Systems.

3.3.31. system_update_cpu_freq

Function	Set CPU frequency. Default is 80 MHz.
Prototype	bool system_update_cpu_freq(uint8 freq)
Parameter	uint8 freq: CPU frequency
	#define SYS_CPU_80MHz 80
	#define SYS_CPU_160MHz 160
Return	true: Success
	false: Failure
Note	System bus frequency is 80 MHz, and it is not affected by CPU frequency. The frequency of UART, SPI, or other peripheral devices, are divided from system bus frequency, so they will not be affected by CPU frequency either.

3.3.32. system_get_cpu_freq

Function	Get CPU frequency.
Prototype	uint8 system_get_cpu_freq(void)
Parameter	none
Return	CPU frequency; unit : MHz.

3.3.33. system_get_flash_size_map

	Get current flash size and flash map.	
	Flash map depends on selection when compiling; for more details please see ESP8266 SDK Getting Started Guide.	

Espressif 23/158 2018.08



```
enum flash_size_map {
                          FLASH\_SIZE\_4M\_MAP\_256\_256 = 0,
                          FLASH_SIZE_2M,
                          FLASH_SIZE_8M_MAP_512_512,
                          FLASH_SIZE_16M_MAP_512_512,
Structure
                          FLASH_SIZE_32M_MAP_512_512,
                          FLASH_SIZE_16M_MAP_1024_1024,
                          FLASH_SIZE_32M_MAP_1024_1024,
                          FLASH_SIZE_64M_MAP_1024_1024,
                          {\sf FLASH\_SIZE\_128M\_MAP\_1024\_1024},
                  };
Prototype
                  enum flash_size_map system_get_flash_size_map(void)
Parameter
 Return
                  flash map
```

Espressif 24/158 2018.08



3.3.34. system_get_rst_info

```
Function
                 Get information about current startup.
                 enum rst_reason {
                        REANSON_DEFAULT_RST = 0, // normal startup by power on
                                              = 1, // hardware watch dog reset
                        REANSON_WDT_RST
                        // exception reset, GPIO status won' t change
                        REANSON_EXCEPTION_RST = 2,
                        // software watch dog reset, GPIO status won' t change
                        REANSON_SOFT_WDT_RST
                                               = 3,
                        // software restart , system_restart , GPIO status won' t
                 change
                        REANSON_SOFT_RESTART
                                               = 4,
                        REANSON_DEEP_SLEEP_AWAKE= 5,
                                                      // wake up from deep-sleep
                        REANSON_EXT_SYS_RST = 6, // external system reset
Structure
                };
                struct rst_info {
                        uint32 reason;
                                                       // enum rst_reason
                        uint32 exccause;
                        uint32 epc1;
                                                       // the address that error
                 occurred
                        uint32 epc2;
                        uint32 epc3;
                        uint32 excvaddr;
                        uint32 depc;
                };
Prototype
                 struct rst_info* system_get_rst_info(void)
Parameter
                none
 Return
                 Information about startup.
```

3.3.35. system_soft_wdt_stop

Function	Stop software watchdog.
Prototype	<pre>void system_soft_wdt_stop(void)</pre>
Parameter	none
Return	none
Note	The software watchdog must not be stopped for too long (over 6 seconds), otherwise it will trigger hardware watchdog reset.



3.3.36. system_soft_wdt_restart

Function	Restart software watchdog.
Prototype	void system_soft_wdt_restart(void)
Parameter	none
Return	none
Note	This API can only be called if software watchdog is stopped (system_soft_wdt_stop).

3.3.37. system_soft_wdt_feed

Function	Feed software watchdog.
Prototype	<pre>void system_soft_wdt_feed(void)</pre>
Parameter	none
Return	none
Note	This API can only be called if software watchdog is enabled.

3.3.38. system_show_malloc

Function	For debugging memory leak issue and printing the memory usage.
Prototype	void system_show_malloc(void)
Parameter	none
Return	none
Note	 To use this API, users need to enable #define MEMLEAK_DEBUG in user_config.h, then refer to the note which is at the beginning of ESP8266_NONOS_SDK\included\mem.h.
	The memory usage which cause memory leak issue may be in the logs, not ensure, just for reference.
	This API is only for debugging. After calling this API, the program may go wrong, so please do not call it in normal usage.



3.3.39. os_memset

Function	Set value of memory.
Prototype	os_memset(void *s, int ch, size_t n)
Parameter	void *s: pointer of memory
	int ch: set value
	size_t n: Size
Return	none
Example	uint8 buffer[32];
	os_memset(buffer, 0, sizeof(buffer));

3.3.40. os_memcpy

Function	Standard function for copying memory content.
Prototype	os_memcpy(void *des, void *src, size_t n)
Parameter	void *des: pointer of destination
	void *src: pointer of source
	size_t n: memory size
Note	<pre>uint8 buffer[4] = {0};</pre>
	os_memcpy(buffer, "abcd", 4);

3.3.41. os_strlen

Function	Get string length.
Prototype	os_strlen(char *s)
Parameter	char *s: string
Return	string length
Example	char *ssid = "ESP8266";
	os_memcpy(softAP_config.ssid, ssid, os_strlen(ssid));

3.3.42. os_printf

Function	Print format.
Prototype	os_printf(const char *s)
Parameter	const char *s: string
Example	os_printf("SDK version: %s \n", system_get_sdk_version());



Note	• Default to be output from UART 0. uart_init in IOT_Demo can set baud rate of UART, and os_install_putc1((void *)uart1_write_char) in it will set os_printf to be output from UART 1.
	Continuously printing more than 125 bytes or repeated calls to this API may cause loss of print data.

3.3.43. os_bzero

Function	Set the first n bytes of string p to be 0, include '\0'.
Prototype	void os_bzero(void *p, size_t n)
Parameter	void *p: pointer of memory need to be set 0
	size_t n: length
Return	none

3.3.44. os_delay_us

Function	Time delay, max : 65535 μs.
Prototype	void os_delay_us(uint16 μs)
Parameter	uint16 μ s: time, unit: μ s
Return	none

3.3.45. os_install_putc1

Function	Register print output function.
Prototype	<pre>void os_install_putc1(void(*p)(char c))</pre>
Parameter	void(*p)(char c): pointer of print function.
Example	os_install_putc1((void *)uart1_write_char) in uart_init will set os_printf to be output from UART 1; otherwise, os_printf default output from UART 0.

3.3.46. os_random

Function	Get a random number.
Prototype	unsigned long os_random(void)
Return	The random number.



3.3.47. os_get_random

Function	Get a random number of specified bytes.
Prototype	int os_get_random(unsigned char *buf, size_t len)
Parameter	unsigned char *buf: pointer of the random number it gets size_t len: specified bytes of the random number
Return	0: Success otherwise: Failure
Example	<pre>int ret = os_get_random((unsigned char *)temp, 7); os_printf("ret %d, value 0x%08x%08x\n\r", ret, temp[1], temp[0]);</pre>

3.3.48. user_rf_cal_sector_set

Function	Set the target flash sector to store RF_CAL parameters.
Prototype	uint32 user_rf_cal_sector_set(void)
Return	The target flash sector to store RF_CAL parameters.
Notes	 The user_rf_cal_sector_set has to be added in application, but need NOT to be called. It will be called inside the SDK. The system parameter area (4 flash sectors) has already been used, so the RF_CAL parameters will be stored in the target sector set by user_rf_cal_sector_set. Since we do not know which sector is available in user data area, users need to set an available sector in the user_rf_cal_sector_set for the SDK to store RF_CAL parameter.
	 If the user_rf_cal_sector_set is not added in the application, the compilation will fail in link stage.
	 Download blank.bin to initialize the sector stored RF_CAL parameter, and download esp_init_data.bin into flash, when the system needs to be initialized, or RF needs to be calibrated again.

Espressif 29/166 2018.08



```
Set the 5th sector from the end of the flash to store the RF_CAL parameter.
            uint32 user_rf_cal_sector_set(void)
                enum flash_size_map size_map = system_get_flash_size_map();
                uint32 rf_cal_sec = 0;
                switch (size_map) {
                    case FLASH_SIZE_4M_MAP_256_256:
                        rf_cal_sec = 128 - 5;
                        break;
                    case FLASH_SIZE_8M_MAP_512_512:
                        rf_cal_sec = 256 - 5;
                        break;
                    case FLASH_SIZE_16M_MAP_512_512:
                    case FLASH_SIZE_16M_MAP_1024_1024:
                        rf_cal_sec = 512 - 5;
Example
                        break;
                    case FLASH_SIZE_32M_MAP_512_512:
                    case FLASH_SIZE_32M_MAP_1024_1024:
                        rf_cal_sec = 512 - 5;
                        break;
                   case FLASH_SIZE_64M_MAP_1024_1024:
                        rf_cal_sec = 2048 - 5;
                        break;
                   case FLASH_SIZE_128M_MAP_1024_1024:
                        rf_cal_sec = 4096 - 5;
                        break;
                   default:
                        rf_cal_sec = 0;
                        break;
                return rf_cal_sec;
```

3.3.49. system_deep_sleep_instant

Function	Configures chip to enter Deep-sleep mode immediately. When the device is in Deep-sleep, it automatically wakes up when time out. Upon waking up, the device boots up from user_init.
Prototype	bool system_deep_sleep_instant(uint64 time_in_us)
Parameter	uint64 time_in_us: the duration of time (µs) when the device is in Deep-sleep.
	The theoretical maximum value of time_in_us can be calculated by formula: (time_in_us / cali) << 12 = 2^32 - 1
	• cali = system_rtc_clock_cali_proc(), the cali is the RTC clock period (in us); bit11 ~ bit0 are decimal. For more details about the cali, please see the API: system_rtc_clock_cali_proc.
	The input value of time_in_us should be less than the theoretical maximum value.
Return	true: Success
	false: Failure



	• Hardware has to support deep-sleep wake up (XPD_DCDC connects to EXT_RSTB with a 0- Ω resistor).
Note	 system_deep_sleep_instant(0): there is no wakeup timer; in order to wake up, connect a GPIO to pin RST; the chip will wake up by a falling-edge on pin RST.
	• After configuration, the chip enters Deep-sleep mode immediately, will not wait till the Wi-Fi core is closed safely. Or you can use <pre>system_deep_sleep</pre> instead.

3.3.50. system_partition_table_regist

Function	Register partition table.
Prototype	<pre>bool system_partition_table_regist(const partition_item_t* partition_table, uint32_t partition_num, uint32_t map)</pre>
Parameter	const partition_item_t* partition_table: the partition table uint32_t partition_num: the partition count uint32_t map: flash map. The value of this parameter should be the same as the flash map selected during the compilation and downloading. Otherwise, it may cause abnormal startup. It's recommended that Marco SPI_FLASH_SIZE_MAP, which stores the value of flash map during the compilation, is passed to this parameter.
Return	true: Success false: Failure
Note	 This API must be called in the user_pre_init to register partition table. If the API returns false, please check your definition of the partition table. Please refer to ESP8266_NONOS_SDK/examples/loT_Demo/user/user_main.c.

3.3.51. system_partition_get_ota_partition_size

Function	Get the size of ota partition.
Prototype	uint32_t system_partition_get_ota_partition_size(void)
Parameter	-
Return	The size of ota partition.
Note	OTA partition is the flash partition where user1.bin or user2.bin is stored.



3.3.52. system_partition_get_item

Function	Get the information of a specific partition.
Prototype	<pre>bool system_partition_get_item(partition_type_t type, partition_item_t* partition_item)</pre>
Parameter	<pre>partition_type_t type: the specific partition type partition_item_t* partition_item: information of the specific partition</pre>
Return	true: Success false: Failure

3.4. SPI Flash Related APIs

SPI flash APIs can be found in: /ESP8266_NONOS_SDK/include/spi_flash.h.

system_param_xxx APIs can be found in: /ESP8266_NONOS_SDK/include/
user_interface.h.

More details about flash read/write operation please see document <u>ESP8266 Flash RW Operation</u>.

3.4.1. spi_flash_get_id

Function	Get ID info of SPI flash.
Prototype	uint32 spi_flash_get_id (void)
Parameter	none
Return	SPI flash ID

3.4.2. spi_flash_erase_sector

Function	Erase sector in flash.
Prototype	SpiFlashOpResult spi_flash_erase_sector (uint16 sec)
Parameter	uint16 sec: Sector number, the count starts at sector 0, 4 KB per sector.
Return	<pre>typedef enum{ SPI_FLASH_RESULT_OK, SPI_FLASH_RESULT_ERR, SPI_FLASH_RESULT_TIMEOUT } SpiFlashOpResult;</pre>

3.4.3. spi_flash_write

Function	Write data to flash. Flash read/write has to be aligned to the 4-byte boundary.
----------	---



Prototype	<pre>SpiFlashOpResult spi_flash_write (uint32 des_addr, uint32 *src_addr, uint32 size)</pre>
Parameter	uint32 des_addr: destination address in flash. uint32 *src_addr: source address of the data. uint32 size: length of data, uint: byte, has to be aligned to the 4-byte boundary.
Return	<pre>typedef enum{ SPI_FLASH_RESULT_OK, SPI_FLASH_RESULT_ERR, SPI_FLASH_RESULT_TIMEOUT } SpiFlashOpResult;</pre>

3.4.4. spi_flash_read

Function	Read data to flash. Flash read/write has to be aligned to the 4-byte boundary.
Prototype	<pre>SpiFlashOpResult spi_flash_read(uint32 src_addr, uint32 * des_addr, uint32 size)</pre>
Parameter	uint32 src_addr: source address in flash. uint32 *des_addr: destination address to keep data. uint32 size: length of data; unit: byte, has to be aligned to the 4-bytes boundary.
Return	<pre>typedef enum { SPI_FLASH_RESULT_OK, SPI_FLASH_RESULT_ERR, SPI_FLASH_RESULT_TIMEOUT } SpiFlashOpResult;</pre>
Example	<pre>uint32 value; uint8 *addr = (uint8 *)&value spi_flash_read(0x3E * SPI_FLASH_SEC_SIZE, (uint32 *)addr, 4); os_printf("0x3E sec:%02x%02x%02x\r\n", addr[0], addr[1], addr[2], addr[3]);</pre>

$3.4.5. \hspace{0.2in} system_param_save_with_protect$

	Write data into flash with protection. Flash read/write has to be aligned to the 4-byte boundary.	
Function	Protection of flash read/write: 3 sectors (4 KB per sector) are used to save 4 KB data with protection; sector 0 and sector 1 are data sectors and back up each other; data is saved alternately; sector 2 is flag sector that points out which sector is keeping the latest data—sector 0 or sector 1.	



Prototype	<pre>bool system_param_save_with_protect (uint16 start_sec, void *param, uint16 len)</pre>
	<pre>uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection.</pre>
Parameter	For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D.
	void *param: pointer of data need to save.
	uint16 len: data length, should less than a sector which is 4*1024.
Return	true: Success
Hetain	false: Failure
	uint32 value;
	uint8 *addr = (uint8 *)&value
Example	<pre>spi_flash_read(0x3E * SPI_FLASH_SEC_SIZE, (uint32 *)addr, 4);</pre>
	os_printf("0x3E sec:%02x%02x%02x%02x\r\n", addr[0], addr[1], addr[2], addr[3]);
Note	For more details about protection of flash read/write, please see ESP8266 Flash RW Operation.

3.4.6. system_param_load

Read protected data from flash. Flash read/write has to be aligned to the 4-byte boundary. Protection of flash read/write: 3 sectors (4 KB per sector) are used to save 4 KB data with protection; sector 0 and sector 1 are data sectors and back up each other; data is saved alternately; sector 2 is flag sector that points out which sector is keeping the latest data—sector 0 or sector 1. bool system_param_load (
data with protection; sector 0 and sector 1 are data sectors and back up each other; data is saved alternately; sector 2 is flag sector that points out which sector is keeping the latest data—sector 0 or sector 1. bool system_param_load (
Prototype uint16 start_sec, uint16 offset, void *param, uint16 len) uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.	Function	data with protection; sector 0 and sector 1 are data sectors and back up each other; data is saved alternately; sector 2 is flag sector that points out which sector
Prototype uint16 offset, void *param, uint16 len uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.		bool system_param_load (
void *param, uint16 len) uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.		uint16 start_sec,
uint16 len) uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.	Prototype	uint16 offset,
uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.		void *param,
 uint16 start_sec: start sector (sector 0) of the 3 sectors which used for flash read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save. 		uint16 len
read/write protection. It cannot be sectors 1 or 2. For example, in IOT_Demo we could use the 3 sectors (3*4 KB) starts from flash 0x3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.)
Ox3D000 for flash read/write protection, so the parameter start_sec should be 0x3D. uint16 offset: offset of data saved in sector. void *param: pointer of data need to save.		
void *param: pointer of data need to save.	Parameter	0x3D000 for flash read/write protection, so the parameter start_sec should be
		uint16 offset: offset of data saved in sector.
uint16 len: data length, should less than a sector which is 4*1024.		void *param: pointer of data need to save.
		uint16 len: data length, should less than a sector which is 4*1024.



Return	true: Success
	false: Failure
Example	uint32 value;
	uint8 *addr = (uint8 *)&value
	<pre>spi_flash_read(0x3E * SPI_FLASH_SEC_SIZE, (uint32 *)addr, 4);</pre>
	os_printf("0x3E sec:%02x%02x%02x%02x\r\n", addr[0], addr[1], addr[2], addr[3]);
Note	For more details about protection of flash read/write, please see <u>ESP8266 Flash</u> <u>RW Operation</u> .

3.4.7. spi_flash_set_read_func

Function	Register user-defined SPI flash read API.
Prototype	void spi_flash_set_read_func (user_spi_flash_read read)
Parameter	user_spi_flash_read: user-defined SPI flash read API.
Return	none
Parameter Definition	<pre>typedef SpiFlashOpResult (*user_spi_flash_read)(SpiFlashChip *spi, uint32 src_addr, uint32 * des_addr, uint32 size)</pre>
Note	This API can be only used in SPI overlap mode, for details please see ESP8266_NONOS_SDK\driver_lib\driver\spi_overlap.c.

3.4.8. spi_flash_erase_protect_enable

Function	Enable SPI flash erase protection.
Note	It can prevent users from erasing the running firmware mistakenly. And users still can erase / write the parameter area on the flash.
Prototype	bool spi_flash_erase_protect_enable(void)
Parameter	none
Return	true: Success false: Failure

3.4.9. spi_flash_erase_protect_disable

Function	Disable SPI flash erase protection.
Prototype	bool spi_flash_erase_protect_disable(void)
Parameter	none



Return	true: Success
	false: Failure

3.5. Wi-Fi Related APIs

Wi-Fi APIs can be found in /ESP8266_NONOS_SDK/include/user_interface.h.

wifi_station_xxx APIs and other APIs which set/get configurations of the ESP8266 Station can only be called if the ESP8266 Station is enabled.

wifi_softap_xxx APIs and other APIs which set/get configurations of the ESP8266 SoftAP can only be called if the ESP8266 SoftAP is enabled.

ESP8266 station supports OPEN, WEP, WPAPSK, WPA2PSK; and encryption AUTO, TKIP, AES, WEP are supported.

ESP8266 softAP supports OPEN, WPAPSK, WPA2PSK; and encryption AUTO, TKIP, AES are supported. But for the group key, only TKIP is supported.

Flash system parameter area is the last 16 KB of flash.

3.5.1. wifi_get_opmode

Function	Get the current operating mode of Wi-Fi.
Prototype	uint8 wifi_get_opmode (void)
Parameter	none
Return	Wi-Fi working modes: • 0x01: Station mode • 0x02: SoftAP mode • 0x03: Station + SoftAP
Note	This API can be only used in SPI overlap mode, for details please see ESP8266_NONOS_SDK\driver_lib\driver\spi_overlap.c.

3.5.2. wifi_get_opmode_default

Function	Get the Wi-Fi operating mode that saved in flash.
Prototype	uint8 wifi_get_opmode_default (void)
Parameter	none
	Wi-Fi working modes:
Return	0x01: Station mode
riotairi	0x02: SoftAP mode
	0x03: Station + SoftAP



	Note	This API can be only used in SPI overlap mode, for details please see ESP8266_NONOS_SDK\driver_lib\driver\spi_overlap.c.
--	------	---

3.5.3. wifi_set_opmode

Function	Set Wi-Fi working mode to Station mode, SoftAP or Station + SoftAP, and save it in flash. The default mode is SoftAP mode.
Prototype	bool wifi_set_opmode (uint8 opmode)
Parameter	 uint8 opmode: Wi-Fi working modes: 0x01: Station mode 0x02: SoftAP mode 0x03: Station + SoftAP
Return	true: Success false: Failure
Note	Versions before ESP8266_NONOS_SDK_V0.9.2, need to call system_restart() after this API; after ESP8266_NONOS_SDK_V0.9.2, need not to restart. This configuration will be saved in flash system parameter area if changed.

3.5.4. wifi_set_opmode_current

Function	Set Wi-Fi working mode to Station mode, SoftAP or Station + SoftAP, and do not update flash.
Prototype	bool wifi_set_opmode_current (uint8 opmode)
Parameter	 uint8 opmode: Wi-Fi working modes: 0x01: Station mode 0x02: SoftAP mode 0x03: Station + SoftAP
Return	true: Success false: Failure

3.5.5. wifi_station_get_config

Function	Get Wi-Fi Station's current configuration.
Prototype	bool wifi_station_get_config (struct station_config *config)
Parameter	struct station_config *config: Wi-Fi Station configuration pointer
Return	true: Success
	false: Failure



3.5.6. wifi_station_get_config_default

Function	Get Wi-Fi Station's configuration from flash memory.
Prototype	bool wifi_station_get_config_default (struct station_config *config)
Parameter	struct station_config *config: Wi-Fi Station configuration pointer
Return	true: Success
	false: Failure

3.5.7. wifi_station_set_config

Function	Set Wi-Fi Station configuration, and save it to flash.
Prototype	bool wifi_station_set_config (struct station_config *config)
Parameter	struct station_config *config: Wi-Fi Station configuration pointer
Return	true: Success false: Failure
Example	<pre>void ICACHE_FLASH_ATTR user_set_station_config(void) {</pre>
Note	 This API can be called only if ESP8266 Station is enabled. If wifi_station_set_config is called in user_init, there is no need to call wifi_station_connect after that, ESP8266 will connect to router automatically; otherwise, wifi_station_connect is needed to connect.
	 In general, station_config.bssid_set need to be 0, otherwise it will check BSSID which is the MAC address of AP. This configuration will be saved in flash system parameter area if changed.



3.5.8. wifi_station_set_config_current

Function	Set Wi-Fi Station configuration; setting in flash is not updated.
Prototype	bool wifi_station_set_config (struct station_config *config)
Parameter	struct station_config *config: Wi-Fi Station configuration pointer
Return	true: Success false: Failure
Example	<pre>void ICACHE_FLASH_ATTR user_set_station_config(void) {</pre>
Note	 This API can be called only if ESP8266 Station is enabled. If wifi_station_set_config is called in user_init, there is no need to call wifi_station_connect after that, ESP8266 will connect to router automatically; otherwise, wifi_station_connect is needed to connect. In general, station_config.bssid_set need to be 0, otherwise it will check BSSID which is the MAC address of AP.

3.5.9. wifi_station_set_cert_key

Function	This API is deprecated; please use wifi_station_set_enterprise_cert_key instead. Set certificate and private key for connecting to WPA2-ENTERPRISE AP.
Prototype	bool wifi_station_set_cert_key (uint8 *client_cert, int client_cert_len, uint8 *private_key, int private_key_len, uint8 *private_key_passwd, int private_key_passwd_len,)



	uint8 *client_cert: certificate; HEX array.
	int client_cert_len: length of certificate.
	uint8 *private_key: private key; HEX array; can NOT be longer than 2048 bits.
Parameter	int private_key_len: length of private key; less than 2048 bits.
	uint8 *private_key_passwd: password for private key; to be supported; can only be NULL now.
	<pre>int private_key_passwd_len: length of password; to be supported; can only be 0 now.</pre>
Return	0: Success
netani	otherwise: Failure
	For example, the private key is BEGIN PRIVATE KEY
Example	Then the array should be uint8 key[]={0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, 0x00 };
	It is the ASCII of the characters, and the array needs to terminate with 0x00.
	• Connecting to WPA2-ENTERPRISE AP needs more than 26 KB of memory, please ensure enough space (system_get_free_heap_size).
	So far, WPA2-ENTERPRISE can only support unencrypted certificate and private key, and only in PEM format.
	- Header of certificate: BEGIN CERTIFICATE
Note	- Header of private key: BEGIN RSA PRIVATE KEY or BEGIN PRIVATE KEY
Note	Please call this API to set certificate and private key before connecting to WPA2-ENTERPRISE AP and the application needs to hold the certificate and private key. Call wifi_station_clear_cert_key to release resources and clear status after connected to the target AP, and then the application can release the certificate and private key.
	If the private key is encrypted, please use OpenSSL PKey command to change it to unencrypted file to use, or use OpenSSI RSA related commands to change it (or change the start TAG).

3.5.10. wifi_station_clear_cert_key

Function	This API is deprecated, please use wifi_station_clear_enterprise_cert_key instead. Release certificate and private key resources and clear related status after connected to the WPA2-ENTERPRISE AP.
Prototype	void wifi_station_clear_cert_key (void)
Parameter	none
Return	none



3.5.11. wifi_station_set_username

Function	This API is deprecated, please use wifi_station_clear_enterprise_cert_key instead.
	Set ESP8266 Station's user name for connecting to WPA2-ENTERPRISE AP.
Prototype	int wifi_station_set_username (uint8 *username, int len)
Parameter	uint8 *username: the user name.
	int len: length of user name.
Return	0 : Success
	otherwise: Failure

3.5.12. wifi_station_clear_username

Function	This API is deprecated, please use wifi_station_clear_enterprise_username instead. Release the user name resources and clear related status after connected to the WPA2-ENTERPRISE AP.
Prototype	void wifi_station_clear_username (void)
Parameter	none
Return	none

3.5.13. wifi_station_connect

Function	Connect Wi-Fi Station to AP.
Prototype	void wifi_station_clear_username (void)
Parameter	none
Return	true: Success false: Failure
Note	If the ESP8266 is already connected to a router, wifi_station_disconnect must be called first, before calling wifi_station_connect.
	Do not call this API in this API need to be called after system initializes and the ESP8266 Station mode is enabled.

3.5.14. wifi_station_disconnect

Function	Disconnects Wi-Fi Station from AP.
Prototype	void wifi_station_clear_username (void)
Parameter	none

Espressif 41/166 2018.08



Return	true: Success false: Failure
Note	Do not call this API in user_init. This API need to be called after system initializes and the ESP8266 Station mode is enabled.

3.5.15. wifi_station_get_connect_status

Function	Get Wi-Fi connection status of ESP8266 Station to AP.
Prototype	uint8 wifi_station_get_connect_status (void)
Parameter	none
Return	<pre>enum{ STATION_IDLE = 0, STATION_CONNECTING, STATION_WRONG_PASSWORD, STATION_NO_AP_FOUND, STATION_CONNECT_FAIL, STATION_GOT_IP };</pre>
Note	In a special case, if you call wifi_station_set_reconnect_policy to disable reconnection, and do not call wifi_set_event_handler_cb to register Wi-Fi event handler, wifi_station_get_connect_status will become invalid and can not get the right status.

3.5.16. wifi_station_scan

Function	Scan all available APs.
Prototype	<pre>bool wifi_station_scan (struct scan_config *config, scan_done_cb_t cb);</pre>
Structure	<pre>struct scan_config { uint8 *ssid;</pre>
Parameter	 struct scan_config *config: AP config for scan. If config==null, scan all APs. If config.ssid==null && config.bssid==null && config.channel!=null, ESP8266 Station interface will scan the APs in a specific channel. If config.ssid!=null && config.bssid==null && config.channel==null, ESP8266 Station will scan the APs with a specific SSID in all channels. scan_done_cb_t cb: callback function after scanning.



Return	true: Success false: Failure
Note	Do not call this API in user_init. This API need to be called after system initializes and the ESP8266 Station mode is enabled.

3.5.17. scan_done_cb_t

Function	Callback function for wifi_station_scan.
Prototype	void scan_done_cb_t (void *arg, STATUS status)
Parameter	<pre>void *arg: information of APs that were found, refer to struct bss_info. STATUS status: get status.</pre>
Return	none
Example	<pre>wifi_station_scan(&config, scan_done); static void ICACHE_FLASH_ATTR scan_done(void *arg, STATUS status) { if (status == 0K) { struct bss_info *bss_link = (struct bss_info *)arg; } }</pre>

3.5.18. wifi_station_ap_number_set

Function	Sets the number of APs that will be cached for ESP8266 Station mode. Whenever ESP8266 Station connects to an AP, it caches a record of this AP's SSID and password. The cached ID index starts from 0.
Prototype	bool wifi_station_ap_number_set (uint8 ap_number)
Parameter	uint8 ap_number: the number of APs that can be recorded (MAX: 5)
Return	true: Success false: Failure
Example	<pre>wifi_station_scan(&config, scan_done); static void ICACHE_FLASH_ATTR scan_done(void *arg, STATUS status) { if (status == OK) { struct bss_info *bss_link = (struct bss_info *)arg; } }</pre>
Note	This configuration will be saved in flash system parameter area if changed.

3.5.19. wifi_station_get_ap_info

Function	Get information of APs recorded by ESP8266 Station.
Prototype	uint8 wifi_station_get_ap_info(struct station_config config[])
Parameter	struct station_config config[]: information of APs; array size has to be 5.



Return	The number of APs recorded.
Example	<pre>struct station_config config[5]; int i = wifi_station_get_ap_info(config);</pre>

3.5.20. wifi_station_ap_change

Function	Switch ESP8266 Station connection to AP as specified.
Prototype	bool wifi_station_ap_change (uint8 new_ap_id)
Parameter	uint8 new_ap_id: AP's record ID; start counting from 0.
Return	true: Success
	false: Failure

3.5.21. wifi_station_get_current_ap_id

Function	Get the current cached ID of AP. ESP8266 records the ID of each AP it connects with. The ID number starts from 0.
Prototype	uint8 wifi_station_get_current_ap_id ();
Parameter	none
Return	The ID of the AP to which ESP8266 is currently connected, in the cached AP list.

3.5.22. wifi_station_get_auto_connect

Function	Check if ESP8266 Station will connect to AP (whose ID is cached) automatically or not when it is powered on.
Prototype	uint8 wifi_station_get_auto_connect(void)
Parameter	none
Return	0: will not connect to AP automatically; otherwise: will connect to AP automatically.

3.5.23. wifi_station_set_auto_connect

Function	Set the ESP8266 Station to connect to the AP (whose ID is cached) automatically or not when powered on. Auto-connection is enabled by default.
Prototype	bool wifi_station_set_auto_connect(uint8 set)
Parameter	 uint8 set: Automatically connect or not: 0: will not connect automatically 1: to connect automatically
Return	true: Success false: Failure

Espressif 44/166 2018.08



Note	Setting the ESP8266 Station to connect to the AP (whose ID is cached) automatically or not when powered on. Auto-connection is enabled by default.
------	--

3.5.24. wifi_station_dhcpc_start

Function	Enable ESP8266 Station DHCP client.
Prototype	bool wifi_station_dhcpc_start(void)
Parameter	none
Return	true: Success false: Failure
Note	 DHCP is enabled by default. This configuration interacts with static IP API (wifi_set_ip_info): If DHCP is enabled, static IP will be disabled; If static IP is enabled, DHCP will be disabled; These settings depend on the last configuration.

3.5.25. wifi_station_dhcpc_stop

Function	Disable ESP8266 Station DHCP client.
Prototype	bool wifi_station_dhcpc_stop(void)
Parameter	none
Return	true: Success
	false: Failure
Note	DHCP is enabled by default.

3.5.26. wifi_station_dhcpc_status

Function	Get ESP8266 Station DHCP client status.
Prototype	enum dhcp_status wifi_station_dhcpc_status(void)
Parameter	none
Return	<pre>enum dhcp_status { DHCP_STOPPED, DHCP_STARTED };</pre>

3.5.27. wifi_station_dhcpc_set_maxtry

Function	Set the maximum number that ESP8266 Station DHCP client will try to reconnect to the AP.
----------	--



Prototype	bool wifi_station_dhcpc_set_maxtry(uint8 num)
Parameter	uint8 num: the maximum number count.
Return	true: Success
	false: Failure

3.5.28. wifi_station_set_reconnect_policy

Function	Set whether the ESP8266 will attempt to reconnect to an AP if disconnected or failed to connect.
Prototype	bool wifi_station_set_reconnect_policy(bool set)
Parameter	bool set: true, enable reconnection; false, disable reconnection.
Return	true: Success false: Failure
Note	It is recommended that the API be called from user_init.This API can only be called when the ESP8266 Station is enabled.

3.5.29. wifi_station_get_rssi

Function	Get RSSI of the AP to which the ESP8266 is connected.
Prototype	sint8 wifi_station_get_rssi(void)
Parameter	none
Return	31: Failure, invalid value.
	other : Success, value of RSSI, in general, RSSI value < 10

3.5.30. wifi_station_set_hostname

Function	Set ESP8266 Station DHCP hostname.
Prototype	bool wifi_station_set_hostname(char* hostname)
Parameter	char* hostname: hostname, max length: 32 characters.
Return	true: Success
	false: Failure



3.5.31. wifi_station_get_hostname

Function	Get ESP8266 Station DHCP hostname.
Prototype	char* wifi_station_get_hostname(void)
Parameter	none
Return	hostname

3.5.32. wifi_softap_get_config

Function	Get the current configuration of Wi-Fi SoftAP.
Prototype	<pre>bool wifi_softap_get_config(struct softap_config *config)</pre>
Parameter	struct softap_config *config: ESP8266 SoftAP configuration information.
Return	true: Success
	false: Failure

3.5.33. wifi_softap_get_config_default

Function	Get Wi-Fi SoftAP configuration saved in flash.
Prototype	bool wifi_softap_get_config_default(struct softap_config *config)
Parameter	struct softap_config *config: ESP8266 SoftAP configuration information.
Return	true: Success
	false: Failure

3.5.34. wifi_softap_set_config

Function	Set Wi-Fi SoftAP configuration and save it in flash.
Prototype	bool wifi_softap_set_config (struct softap_config *config)
Parameter	struct softap_config *config: Wi-Fi SoftAP configuration pointer
Return	true: Success false: Failure
Note	 This API can be called only if the ESP8266 SoftAP is enabled. This configuration will be saved in flash system parameter area if changed. In SoftAP + Station mode, the ESP8266 SoftAP will adjust its channel configuration to be the as same as that of the ESP8266 Station. More details please see <i>Appendix.A</i>.

Espressif 47/166 2018.08



3.5.35. wifi_softap_set_config_current

Function	Set Wi-Fi SoftAP configuration; settings are not updated in flash memory.
Prototype	bool wifi_softap_set_config_current (struct softap_config *config)
Parameter	struct softap_config *config: Wi-Fi SoftAP configuration pointer
Return	true: Success false: Failure
Note	 This API can be called only if the ESP8266 SoftAP is enabled. In SoftAP + Station mode, the ESP8266 SoftAP will adjust its channel configuration to be the as same as that of the ESP8266 Station. For more details please see <i>Appendix A</i>.

3.5.36. wifi_softap_get_station_num

Function	Count the number of Stations connected to the ESP8266 SoftAP.
Prototype	uint8 wifi_softap_get_station_num(void)
Parameter	none
Return	Number of Stations connected to ESP8266 SoftAP.

3.5.37. wifi_softap_get_station_info

Function	Get information on connected Station devices under SoftAP mode, including MAC and IP.
Prototype	struct station_info * wifi_softap_get_station_info(void)
Parameter	none
Return	struct station_info*: station information structure.
Note	This API depends on DHCP, so it cannot get static IP, etc. in case DHCP is not used.

3.5.38. wifi_softap_free_station_info

Function	Frees the struct station_info by calling the wifi_softap_get_station_info function.
Prototype	void wifi_softap_free_station_info(void)
Parameter	none
Return	none

Espressif 48/166 2018.08



```
Example 1 (getting MAC and IP information):
                struct station_info * station = wifi_softap_get_station_info();
                struct station_info * next_station;
                while(station) {
                    os_printf(bssid : MACSTR, ip : IPSTR/n,
                            MAC2STR(station->bssid), IP2STR(&station->ip));
                    next_station = STAILQ_NEXT(station, next);
                                                   // Free it directly
                    os_free(station);
                    station = next_station;
Example
                }
                Example 2 (getting MAC and IP information):
                struct station_info * station = wifi_softap_get_station_info();
                while(station){
                    os_printf(bssid : MACSTR, ip : IPSTR/n,
                            MAC2STR(station->bssid), IP2STR(&station->ip));
                    station = STAILQ_NEXT(station, next);
                wifi_softap_free_station_info(); // Free it by calling functions
```

3.5.39. wifi_softap_dhcps_start

Function	Enable ESP8266 SoftAP DHCP server.
Prototype	bool wifi_softap_dhcps_start(void)
Parameter	none
Return	true: Success
	false: Failure
	DHCP is enabled by default.
	This configuration interacts with static IP API (wifi_set_ip_info):
Note	- If DHCP is enabled, static IP will be disabled;
	- If static IP is enabled, DHCP will be disabled;
	- These settings depend on the last configuration.

3.5.40. wifi_softap_dhcps_stop

Function	Disable ESP8266 SoftAP DHCP server.
Prototype	bool wifi_softap_dhcps_stop(void)
Parameter	none
Return	true: Success
	false: Failure
Note	DHCP is enabled by default.



3.5.41. wifi_softap_set_dhcps_lease

Function	Set the IP range that can be allocated by the ESP8266 SoftAP DHCP server.
Prototype	bool wifi_softap_set_dhcps_lease(struct dhcps_lease *please)
Parameter	<pre>struct dhcps_lease { struct ip_addr start_ip; struct ip_addr end_ip; };</pre>
Poturn	true: Success
Return	false: Failure
Example	<pre>void dhcps_lease_test(void) { struct dhcps_lease dhcp_lease; const char* start_ip = "192.168.5.100"; const char* end_ip = "192.168.5.105"; dhcp_lease.start_ip.addr = ipaddr_addr(start_ip); dhcp_lease.end_ip.addr = ipaddr_addr(end_ip); wifi_softap_set_dhcps_lease(&dhcp_lease); } or void dhcps_lease_test(void) { struct dhcps_lease dhcp_lease; IP4_ADDR(&dhcp_lease.start_ip, 192, 168, 5, 100); IP4_ADDR(&dhcp_lease.end_ip, 192, 168, 5, 105); wifi_softap_set_dhcps_lease(&dhcp_lease); } void user_init(void) { struct ip_info info; wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode wifi_softap_dhcps_stop(); IP4_ADDR(&info.ip, 192, 168, 5, 1); IP4_ADDR(&info.ip, 192, 168, 5, 1); IP4_ADDR(&info.netmask, 255, 255, 255, 0); wifi_set_ip_info(SOFTAP_IF, &info); dhcps_lease_test(); wifi_softap_dhcps_start(); } </pre>
Note	 IP range has to be in the same sub-net with the ESP8266 SoftAP IP address. This API can only be called when DHCP server is disabled (wifi_softap_dhcps_stop). This configuration only takes effect on next wifi_ SoftAP_dhcps_start; if then wifi_softap_dhcps_stop is called, users need to call this API to set IP range again if needed, and then call wifi_softap_dhcps_start for the configuration to take effect.



3.5.42. wifi_softap_get_dhcps_lease

Function	Query the IP range that can be allocated by the ESP8266 SoftAP DHCP server.
Prototype	bool wifi_softap_get_dhcps_lease(struct dhcps_lease *please)
Return	true: Success
	false: Failure
Note	This API can only be called when ESP8266 SoftAP DHCP server is enabled.

3.5.43. wifi_softap_set_dhcps_lease_time

Function	Set ESP8266 SoftAP DHCP server lease time, 120 minutes by default.
Prototype	bool wifi_softap_set_dhcps_lease_time(uint32 minute)
Parameter	uint32 minute: lease time, uint: minute, range: [1, 2880].
Return	true: Success
	false: Failure
Note	This API can only be called when ESP8266 SoftAP DHCP server is enabled.

3.5.44. wifi_softap_get_dhcps_lease_time

Function	Get ESP8266 SoftAP DHCP server lease time.
Prototype	uint32 wifi_softap_get_dhcps_lease_time(void)
Return	Lease time; uint: minute.
Note	This API can only be called when ESP8266 SoftAP DHCP server is enabled.

3.5.45. wifi_softap_reset_dhcps_lease_time

Function	Reset ESP8266 SoftAP DHCP server lease time to its default value, which is 120 minutes.
Prototype	bool wifi_softap_reset_dhcps_lease_time(void)
Return	true: Success
	false: Failure
Note	This API can only be called when ESP8266 SoftAP DHCP server is enabled.

Espressif 51/166 2018.08



3.5.46. wifi_softap_dhcps_status

Function	Get ESP8266 SoftAP DHCP server status.
Prototype	enum dhcp_status wifi_softap_dhcps_status(void)
Parameter	none
Return	<pre>enum dhcp_status { DHCP_STOPPED, DHCP_STARTED };</pre>
Note	This API can only be called when ESP8266 SoftAP DHCP server is enabled.

3.5.47. wifi_softap_set_dhcps_offer_option

Function	Set ESP8266 SoftAP DHCP server option.
Structure	<pre>enum dhcps_offer_option{ OFFER_START = 0x00, OFFER_ROUTER = 0x01, OFFER_END };</pre>
Prototype	bool wifi_softap_set_dhcps_offer_option(uint8 level, void* optarg)
Parameter	 uint8 level: OFFER_ROUTER set router option void* optarg: enabled by default bit0, 0 disables router information from ESP8266 SoftAP DHCP server; bit0, 1 enables router information from ESP8266 SoftAP DHCP server.
Return	true: Success false: Failure
Example	<pre>uint8 mode = 0; wifi_softap_set_dhcps_offer_option(OFFER_ROUTER, &mode);</pre>

3.5.48. wifi_set_phy_mode

Function	Set ESP8266 physical mode (802.11b/g/n).
Prototype	bool wifi_set_phy_mode(enum phy_mode mode)
Parameter	<pre>enum phy_mode mode : physical mode enum phy_mode { PHY_MODE_11B = 1, PHY_MODE_11G = 2, PHY_MODE_11N = 3 };</pre>



Return	true: Success
	false: Failure
Note	ESP8266 SoftAP only support 802.11b/g.
	Users can set to be 802.11g mode for consumption.

3.5.49. wifi_get_phy_mode

Function	Get ESP8266 physical mode (802.11b/g/n).
Prototype	enum phy_mode wifi_get_phy_mode(void)
Parameter	none
Return	<pre>enum phy_mode{ PHY_MODE_11B = 1, PHY_MODE_11G = 2, PHY_MODE_11N = 3 };</pre>

3.5.50. wifi_get_ip_info

Function	Get IP info of Wi-Fi Station or SoftAP interface.
Prototype	<pre>bool wifi_get_ip_info(uint8 if_index, struct ip_info *info)</pre>
Parameter	uint8 if_index: the interface to get IP info: 0x00 for STATION_IF; 0x01 for SOFTAP_IF. struct ip_info *info: pointer to get IP info of a certain interface.
Return	true: Success false: Failure
Note	This API is available after initialization, do not call it in user_init.

3.5.51. wifi_set_ip_info

Function	Set the IP address of ESP8266 Station or SoftAP.
Prototype	<pre>bool wifi_set_ip_info(uint8 if_index, struct ip_info *info)</pre>
Parameter	<pre>uint8 if_index: set Station IP or SoftAP IP. #define STATION_IF</pre>



Return	true: Success
	false: Failure
Example	<pre>wifi_set_opmode(STATIONAP_MODE); //Set softAP + station mode struct ip_info info; wifi_station_dhcpc_stop(); wifi_softap_dhcps_stop(); IP4_ADDR(&info.ip, 192, 168, 3, 200); IP4_ADDR(&info.gw, 192, 168, 3, 1); IP4_ADDR(&info.netmask, 255, 255, 255, 0); wifi_set_ip_info(STATION_IF, &info); IP4_ADDR(&info.ip, 10, 10, 10, 1); IP4_ADDR(&info.gw, 10, 10, 10, 1); IP4_ADDR(&info.netmask, 255, 255, 255, 0); wifi_set_ip_info(SOFTAP_IF, &info); wifi_softap_dhcps_start();</pre>
Note	To set static IP, please disable DHCP first (wifi_station_dhcpc_stop or wifi_softap_dhcps_stop): • If static IP is enabled, DHCP will be disabled; • If DHCP is enabled, static IP will be disabled.

3.5.52. wifi_set_macaddr

Function	Set MAC address.
Prototype	<pre>bool wifi_set_macaddr(uint8 if_index, uint8 *macaddr)</pre>
Parameter	uint8 if_index: set station MAC or SoftAP MAC. #define STATION_IF
Return	true: Success false: Failure
Example	<pre>wifi_set_opmode(STATIONAP_MODE); char sofap_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab}; char sta_mac[6] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xab}; wifi_set_macaddr(SOFTAP_IF, sofap_mac); wifi_set_macaddr(STATION_IF, sta_mac);</pre>



Note	This API can only be called in user_init.
	ESP8266 SoftAP and station have different MAC addresses, please do not set them to be the same.
	The bit 0 of the first byte of ESP8266 MAC address can not be 1. For example, MAC address can be "1a:XX:XX:XX:XX:XX", but can not be "15:XX:XX:XX:XX". The bit 0 of the first byte of ESP8266 MAC address can not be 1. For example, MAC address can be "1a:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX

3.5.53. wifi_get_macaddr

Function	Get MAC address.
Prototype	<pre>bool wifi_get_macaddr(uint8 if_index, uint8 *macaddr)</pre>
Parameter	uint8 if_index: get station MAC or SoftAP MAC. #define STATION_IF
Return	true: Success false: Failure

3.5.54. wifi_set_sleep_type

Function	Sets sleep type for power saving. Set NONE_SLEEP_T to disable power saving.
Prototype	bool wifi_set_sleep_type(enum sleep_type type)
Parameter	enum sleep_type type: sleep type.
Return	true: Success false: Failure
Note	Default mode: Modem-sleep. In order to lower the power comsumption, ESP8266 changes the TCP timer tick from 250ms to 3s in light-sleep mode, which leads to increased timeout for TCP timer. Therefore, the modem-sleep or deep-sleep mode should be used where there is a requirement for the accurancy of the TCP timer.



3.5.55. wifi_get_sleep_type

Function	Get sleep type.
Prototype	enum sleep_type wifi_get_sleep_type(void)
Parameter	none
Return	<pre>enum sleep_type { NONE_SLEEP_T = 0; LIGHT_SLEEP_T, MODEM_SLEEP_T };</pre>

3.5.56. wifi_status_led_install

Function	Install Wi-Fi status LED.
Prototype	<pre>void wifi_status_led_install (uint8 gpio_id, uint32 gpio_name, uint8 gpio_func)</pre>
Parameter	uint8 gpio_id: GPIO number. uint8 gpio_name: GPIO mux name. uint8 gpio_func: GPIO function.
Return	none
Example	Use GPI00 as WiFi status LED #define HUMITURE_WIFI_LED_IO_MUX PERIPHS_IO_MUX_GPI00_U #define HUMITURE_WIFI_LED_IO_NUM 0 #define HUMITURE_WIFI_LED_IO_FUNC FUNC_GPI00 wifi_status_led_install(HUMITURE_WIFI_LED_IO_NUM,

3.5.57. wifi_status_led_uninstall

Function	Uninstall Wi-Fi status LED.
Prototype	void wifi_status_led_uninstall ()
Parameter	none
Return	none



3.5.58. wifi_set_broadcast_if

Function	Set ESP8266 to send UDP broadcast from Station interface or SoftAP interface, or both Station and SoftAP interfaces. The default interface is SoftAP.
Prototype	bool wifi_set_broadcast_if (uint8 interface)
Parameter	uint8 interface:
	• 1: Station
	• 2: SoftAP
	• 3: Station + SoftAP
Return	true: Success
	false: Failure
Note	If broadcast is set to be in station interface only, the SoftAP DHCP server will be disabled.

3.5.59. wifi_get_broadcast_if

Function	Get interface which ESP8266 sends UDP broadcast from. This is usually used when you have Station + SoftAP mode to avoid ambiguity.
Prototype	uint8 wifi_get_broadcast_if (void)
Parameter	none
	1: Station
Return	2: SoftAP
	3: Both Station and SoftAP

Espressif 57/166 2018.08



3.5.60. wifi_set_event_handler_cb

Function	Register Wi-Fi event handler.
Prototype	void wifi_set_event_handler_cb(wifi_event_handler_cb_t cb)
Parameter	wifi_event_handler_cb_t cb: callback
Return	none
Example	<pre>void wifi_handle_event_cb(System_Event_t *evt) { os_printf("event %x\n", evt->event); switch (evt->event) { case EVENT_STAMODE_CONNECTED:</pre>



3.5.61. wifi_wps_enable

Function	Enable Wi-Fi WPS function.
Structure	<pre>typedef enum wps_type { WPS_TYPE_DISABLE=0, WPS_TYPE_PBC, WPS_TYPE_PIN, WPS_TYPE_DISPLAY, WPS_TYPE_MAX, }WPS_TYPE_HAX,</pre>
Prototype	bool wifi_wps_enable(WPS_TYPE_t wps_type)
Parameter	WPS_TYPE_t wps_type: WPS type; so far only WPS_TYPE_PBC is supported.
Return	true: Success false: Failure
Note	WPS can only be used when ESP8266 Station is enabled.

3.5.62. wifi_wps_disable

Function	Disable Wi-Fi WPS function and release resources allocated to it.
Prototype	bool wifi_wps_disable(void)
Parameter	none
Return	true: Success
	false: Failure

3.5.63. wifi_wps_start

Function	WPS starts.
Prototype	bool wifi_wps_start(void)
Parameter	none
Return	true: WPS starts. But it does not mean Wi-Fi protection setup is successfully completed. false: Failure



3.5.64. wifi_set_wps_cb

Function	Set WPS callback; parameter of the callback is the status of WPS.
Callback and Parameter Struture	<pre>typedef void (*wps_st_cb_t)(int status); enum wps_cb_status { WPS_CB_ST_SUCCESS = 0, WPS_CB_ST_FAILED, WPS_CB_ST_TIMEOUT, WPS_CB_ST_WEP, // WPS failed because that WEP is not supported WPS_CB_ST_SCAN_ERR, // can not find the target WPS AP</pre>
Prototype	bool wifi_set_wps_cb(wps_st_cb_t cb)
Parameter	wps_st_cb_t cb: callback
Return	true: Success false: Failure
Note	 If parameter status == WPS_CB_ST_SUCCESS in WPS callback, it means WPS got AP's information, user can call wifi_wps_disable to disable WPS and release resource, then call wifi_station_connect to connect to target AP. Otherwise, it means that WPS failed, user can create a timer to retry WPS by wifi_wps_start after a while, or call wifi_wps_disable to disable WPS and release resource.



${\it 3.5.65. wifi_register_send_pkt_freedom_cb}$

Function	Register a callback for sending user-defined 802.11 packets.
Callback Definition	typedef void (*freedom_outside_cb_t)(uint8 status); parameter status: 0 — packet sending succeeds; otherwise — failed. The send callback can determine the status of a transmitted packet, however, note that: • For unicast packet: - If the status is OK in the freedom_outside_cb_t, but the target device failed to receive the packet, the reasons may be as follows: 1. It may have been corrupted by other unexpected devices. 2. Incorrect key for communication. 3. the application is lost or missed the packet. Solution: handshake mechanism should be used in application to get a high success rate with packet transmission. - If the status is FAIL in the freedom_outside_cb_t, and the target device does receive the packet, the possible reason could be that the sender fails to receive the ACK because of busy channel. Solution: the sender application should re-transmit the packet, and the receiver should detect the retransmitted packet. • For multicast packet (include broadcast packet): - If the status is OK in the freedom_outside_cb_t, it indicates that the packet is sent successfully. - If the status is FAIL in the freedom_outside_cb_t, it indicates that
Prototype	sending failed.
Parameter	<pre>int wifi_register_send_pkt_freedom_cb(freedom_outside_cb_t cb)</pre>
Parameter	freedom_outside_cb_t cb: callback
Return	0: Success -1: Failure
Note	Only after the previous packet was sent, entered the freedom_outside_cb_t, the next packet is allowed to be sent.

3.5.66. wifi_unregister_send_pkt_freedom_cb

Function	Unregister the callback for sending packets freedom.
Prototype	void wifi_unregister_send_pkt_freedom_cb(void)
Parameter	none
Return	none



3.5.67. wifi_send_pkt_freedom

Function	Send user-defined 802.11 packets.
Prototype	int wifi_send_pkt_freedom(uint8 *buf, int len,bool sys_seq)
	uint8 *buf: pointer of packet
Parameter	int len: packet length
	bool sys_seq: follow the system's 802.11 packets sequence number or not; if it is true, the sequence number will be increased 1 every time a packet is sent.
Return	0: Success
rictarri	-1: Failure
	Packet has to be the whole 802.11 packet, excluding the FCS. The length of the packet has to be longer than the minimum length of the header of 802.11 packet, which is 24 bytes, and less than 1400 bytes.
	Duration area is invalid for user, it will be filled in SDK.
Note	The rate of sending packet is same as the management packet which is the same as the system rate of sending packets.
	Can send: unencrypted data packet, unencrypted beacon/probe req/probe resp.
	Can NOT send: all encrypted packets (the encrypt bit in the packet has to be 0, otherwise it is not supported), control packet, other management packet except unencrypted beacon/probe req/probe resp.
	Only after the previous packet was sent, and the sent callback is entered, the next packet is allowed to send. Otherwise, wifi_send_pkt_freedom will return "fail".

3.5.68. wifi_rfid_locp_recv_open

Function	Enable RFID LOCP (Location Control Protocol) to receive WDS packets.
Prototype	int wifi_rfid_locp_recv_open(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.5.69. wifi_rfid_locp_recv_close

Function	Disable RFID LOCP (Location Control Protocol).
Prototype	void wifi_rfid_locp_recv_close(void)
Parameter	none
Return	none



3.5.70. wifi_register_rfid_locp_recv_cb

Function	Register a callback on receiving WDS packets, only if the first MAC address of the WDS packet is a multicast address.
Callback Definition	<pre>typedef void (*rfid_locp_cb_t)(uint8 *frm, int len, int rssi);</pre>
	uint8 *frm: point to the head of 802.11 packet.
Parameter	int len: packet length.
	int rssi: signal strength.
Prototype	<pre>int wifi_register_rfid_locp_recv_cb(rfid_locp_cb_t cb)</pre>
Return	0: Success
11040111	otherwise: Failure

3.5.71. wifi_unregister_rfid_locp_recv_cb

Function	Unregister the callback of receiving WDS packets.
Prototype	void wifi_unregister_rfid_locp_recv_cb(void)
Parameter	none
Return	none

3.5.72. wifi_enable_gpio_wakeup

Function	Set a GPIO to wake the ESP8266 up from light-sleep mode.
Prototype	<pre>void wifi_enable_gpio_wakeup(uint32 i, GPIO_INT_TYPE intr_status)</pre>
Parameter	uint32 i: GPIO number, range: [0, 15].
	GPIO_INT_TYPE intr_status: status of GPIO interrupt to trigger the wakeup process.
Return	none
	ESP8266 will be wakened from Light-sleep, when the GPIO12 is in low-level.
Example	GPIO_DIS_OUTPUT(12);
Example	PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPI012);
	<pre>wifi_enable_gpio_wakeup(12, GPIO_PIN_INTR_LOLEVEL);</pre>
Notes	If the ESP8266 enters light-sleep automatically (wifi_set_sleep_type(LIGHT_SLEEP_T);), after being waken up by GPIO, when the chip attempts to sleep again, it will check the status of the GPIO:
	If the GPIO is still in the wakeup status, the EP8266 will enter modem-sleep mode instead;
	If the GPIO is NOT in the wakeup status, the ESP8266 will enter light-sleep mode.



3.5.73. wifi_disable_gpio_wakeup

Function	Disable the function that the GPIO can wake the ESP8266 up from light-sleep mode.
Prototype	void wifi_disable_gpio_wakeup(void)
Parameter	none
Return	none

3.5.74. wifi_set_country

Function	Set WiFi country information.
Prototype	bool wifi_set_country(wifi_country_t *country)
Parameter	wifi_country_t *country: the WiFi country
Notes	The default WiFi country code is {.cc="CN", .schan=1, .nchan=13, policy=WIFI_COUNTRY_POLICY_AUTO}
	If policy=WIFI_COUNTRY_POLICY_AUTO, the country code of ESP8266 station will change automatically to be the same as the AP that ESP is connected to; and when the WiFi connection ends, its country configuration will change back to the original one.
	If policy=WIFI_COUNTRY_POLICY_MANUAL, the country code of ESP8266 will always be the configured one.
	If the ESP8266 is in station+softAP mode, the softAP's country IE in the probe response/beacon will always be the same as the station's country info.
	The country configuration will not be saved into flash.
Return	true: Success
netum	false: Failure

3.5.75. wifi_get_country

Function	Get WiFi country information.
Prototype	bool wifi_get_country(wifi_country_t *country)
Parameter	wifi_country_t *country: the WiFi country
Return	true: Success
	false: Failure

Espressif 64/166 2018.08



3.5.76. wifi_set_sleep_level

Function	Set sleep level of modem sleep and light sleep.
Prototype	bool wifi_set_sleep_level(enum sleep_level level)
Parameter	level - sleep level of modem sleep and light sleep
Return	true: Success false: Failure
Notes	 This configuration should be called before calling wifi_set_sleep_type, if needed. Modem-sleep and light sleep mode have minimum and maximum sleep levels. In minimum sleep level, station wakes up at every DTIM to receive beacon. Broadcast data will not be lost because it is transmitted after DTIM. However, it can not save much more power if DTIM period is short, as specified in AP. In maximum sleep level, station wakes up at every listen interval to receive beacon. Broadcast data may be lost because station may be in sleep state at DTIM time. If listen interval is longer, more power will be saved, but it's very likely to lose more broadcast data. Default setting is minimum sleep level. Listen interval can be configured by calling API wifi_set_listen_interval.

3.5.77. wifi_get_sleep_level

Function	Get sleep level of modem sleep and light sleep.
Prototype	enum sleep_level wifi_get_sleep_level(void)
Parameter	none
Return	true: Success
	false: Failure

3.5.78. wifi_set_listen_interval

Function	Set listen interval of maximum sleep level for modem sleep and light sleep.
Prototype	bool wifi_set_listen_interval(uint8 interval)
Parameter	interval - listen interval of maximum sleep level for modem sleep and light sleep. Unit: one beacon interval of AP. Range: [1, 10]
Return	true: Success
	false: Failure

Espressif 65/166 2018.08



	It only works when sleep level is set as MAX_SLEEP_T.
	It should be called following the order below:
Notes	- wifi_set_sleep_level(MAX_SLEEP_T)
	- wifi_set_listen_interval
	- wifi_set_sleep_type

$3.5.79.\ wifi_get_listen_interval$

Function	Get listen interval of maximum sleep level for modem sleep and light sleep.
Prototype	uint8 wifi_get_listen_interval(void)
Parameter	none
Return	listen interval of maximum sleep level for modem sleep and light sleep



3.6. Rate Control APIs

Wi-Fi rate control APIs can be found in /ESP8266_NONOS_SDK/include/user_interface.h.

3.6.1. wifi_set_user_fixed_rate

Function	Set the fixed rate and mask of sending data from ESP8266.
Structure and Definition	enum FIXED_RATE { PHY_RATE_48 = 0x8, PHY_RATE_24 = 0x9, PHY_RATE_12 = 0xA, PHY_RATE_6 = 0xB, PHY_RATE_54 = 0xC, PHY_RATE_36 = 0xD, PHY_RATE_18 = 0xE, PHY_RATE_19 = 0xF, } #define FIXED_RATE_MASK_NONE (0x00) #define FIXED_RATE_MASK_STA (0x01) #define FIXED_RATE_MASK_AP (0x02) #define FIXED_RATE_MASK_ALL (0x03)
Prototype	int wifi_set_user_fixed_rate(uint8 enable_mask, uint8 rate)
Parameter	 uint8 enable_mask: 0x00: disable the fixed rate 0x01: use the fixed rate on ESP8266 Station 0x02: use the fixed rate on ESP8266 SoftAP 0x03: use the fixed rate on ESP8266 Station and SoftAP uint8 rate: value of the fixed rate
Return	0: Success otherwise: Failure
Note	 Only if the corresponding bit in enable_mask is 1, ESP8266 Station or SoftAP will send data in the fixed rate. If the enable_mask is 0, both ESP8266 Station and SoftAP will not send data in the fixed rate. ESP8266 Station and SoftAP share the same rate, they can not be set to different data rates.



3.6.2. wifi_get_user_fixed_rate

Function	Get the fixed rate and mask of ESP8266.
Prototype	<pre>int wifi_get_user_fixed_rate(uint8 *enable_mask, uint8 *rate)</pre>
Parameter	uint8 *enable_mask: pointer of the enable_mask.
	uint8 *rate: pointer of the fixed rate.
Return	0: Success
	otherwise: Failure

3.6.3. wifi_set_user_sup_rate

Function	Set the rate range in the IE of support rate in ESP8266's beacon, probe req/resp and other packets. Tell other devices about the rate range supported by ESP8266 to limit the rate of sending packets from other devices.
Parameter	enum support_rate { RATE_11B5M
Prototype	int wifi_set_user_sup_rate(uint8 min, uint8 max)
Parameter	uint8 min: the minimum value of the support rate, according to enum support_rate. uint8 max: the maximum value of the support rate, according to enum support_rate.
Return	0: Success otherwise: Failure
Example	wifi_set_user_sup_rate(RATE_11G6M, RATE_11G24M);
Note	This API can only support 802.11g now, but it will support 802.11b in next version.



3.6.4. wifi_set_user_rate_limit

Function	Limit the initial rate of sending data from ESP8266. The rate of retransmission is not limited by this API.
Parameter Definition	enum RATE_11B_ID { RATE_11B_B1M = 0, RATE_11B_BSM = 1, RATE_11B_BM = 2, RATE_11B_BIM = 3, } enum RATE_11G_ID { RATE_11G_G54M = 0, RATE_11G_G36M = 2, RATE_11G_G24M = 3, RATE_11G_G24M = 4, RATE_11G_G12M = 5, RATE_11G_G12M = 5, RATE_11G_G6M = 7 RATE_11G_BSM = 8, RATE_11G_BSM = 8, RATE_11G_BBM = 10 } enum RATE_11N_ID { RATE_11N_MCS7 = 0, RATE_11N_MCS7 = 1, RATE_11N_MCS6 = 2, RATE_11N_MCS6 = 2, RATE_11N_MCS3 = 5, RATE_11N_MCS3 = 5, RATE_11N_MCS4 = 4, RATE_11N_MCS5 = 6, RATE_11N_MCS5 = 8, RATE_11N_MCS6 = 10, RATE_11N_MCS6
Prototype	<pre>bool wifi_set_user_rate_limit(uint8 mode, uint8 ifidx, uint8 max, uint8 min)</pre>
Parameter	uint8 mode: Wi-Fi mode #define RC_LIMIT_11B
Return	true: Success false: Failure
Example	<pre>// Set the rate limitation of ESP8266 Station in 11G mode, 6M ~ 18M. wifi_set_user_rate_limit(RC_LIMIT_11G, 0, RATE_11G_G18M, RATE_11G_G6M);</pre>



3.6.5. wifi_set_user_limit_rate_mask

Function	Set the interfaces of ESP8266 whose rate of sending packets is limited by wifi_set_user_rate_limit.	
Definition	#define LIMIT_RATE_MASK_NONE (0x00) #define LIMIT_RATE_MASK_STA (0x01) #define LIMIT_RATE_MASK_AP (0x02) #define LIMIT_RATE_MASK_ALL (0x03)	
Prototype	uint8 wifi_get_user_limit_rate_mask(void)	
Parameter	uint8 enable_mask: 0x00 - disable the limitation on both ESP8266 Station and SoftAP; 0x01 - enable the limitation on ESP8266 Station; 0x02 - enable the limitation on ESP8266 SoftAP; 0x03 - enable the limitation on both ESP8266 Station and SoftAP.	
Return	true: Success false: Failure	

3.6.6. wifi_get_user_limit_rate_mask

Function	Get the interfaces of ESP8266 whose rate of sending data is limited by ifi_set_user_rate_limit.	
Prototype	nt8 wifi_get_user_limit_rate_mask(void)	
Parameter	one	
Return	0x00: both ESP8266 Station and SoftAP are not limited.	
	0x01: ESP8266 Station is limited.	
	0x02: ESP8266 SoftAP is limited.	
	0x03: both ESP8266 Station and SoftAP are limited.	



3.7. Force Sleep APIs

Force Sleep APIs can be found in: /ESP8266_NONOS_SDK/include/user_interface.h.

wifi_set_opmode has to be set to NULL_MODE before entering forced sleep mode. Then users need to wake ESP8266 up from sleep, or wait till the sleep time out and enter the wakeup callback (register by wifi_fpm_set_wakeup_cb). Disable the force sleep function by wifi_fpm_close before setting Wi-Fi mode back to normal mode.

Timer will prevent the chip from entering light-sleep mode, please disable all timers in application before entering light-sleep. For more details please see *3.7.9 Example*.

In order to lower the power comsumption, ESP8266 changes the TCP timer tick from 250ms to 3s in light-sleep mode, which leads to increased timeout for TCP timer. Therefore, the modem-sleep or deep-sleep mode should be used where there is a requirement for the accurancy of the TCP timer.

3.7.1. wifi_fpm_open

Function	nable force sleep function.	
Prototype	d wifi_fpm_open (void)	
Parameter	ne	
Return	none	
Default	Force sleep function is disabled.	

3.7.2. wifi_fpm_close

Function	Disable force sleep function.	
Prototype	d wifi_fpm_close (void)	
Parameter	one	
Return	none	

3.7.3. wifi_fpm_do_wakeup

Function	Wake ESP8266 up from MODEM_SLEEP_T force sleep.	
Prototype	oid wifi_fpm_do_wakeup (void)	
Parameter	ne	
Return	none	
Note	This API can only be called when MODEM_SLEEP_T force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.	



3.7.4. wifi_fpm_set_wakeup_cb

Function	eet a wake-up callback function to be called on wake-up from force sleep ecause of timeout.	
Prototype	id wifi_fpm_set_wakeup_cb(void (*fpm_wakeup_cb_func)(void))	
Parameter	oid (*fpm_wakeup_cb_func)(void) : callback on wake-up	
Return	none	
Note	This API can only be called when force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.	
	<pre>fpm_wakeup_cb_func will be called after system wakes up only if the force sleep time out (wifi_fpm_do_sleep and the parameter is not 0xFFFFFFF).</pre>	
	fpm_wakeup_cb_func will not be called if wake-up is caused by wifi_fpm_do_wakeup from MODEM_SLEEP_T type force sleep.	

3.7.5. wifi_fpm_do_sleep

Function	Force ESP8266 to enter sleep mode, and it will wake up automatically on time out.	
Prototype	<pre>int8 wifi_fpm_do_sleep (uint32 sleep_time_in_us)</pre>	
Parameter	uint32 sleep_time_in_us: sleep time, ESP8266 will wake up automatically on time out. Unit: us. Range: 10000 ~ 268435455(0xFFFFFFF)	
	If sleep_time_in_us is 0xFFFFFFF, the ESP8266 will sleep till be woke up as below:	
	If wifi_fpm_set_sleep_type is set to be LIGHT_SLEEP_T, ESP8266 can wake up by GPIO.	
	If wifi_fpm_set_sleep_type is set to be MODEM_SLEEP_T, ESP8266 can wake up by wifi_fpm_do_wakeup.	
	0, setting successful;	
Return	-1, failed to sleep, sleep status error;	
	-2, failed to sleep, force sleep function is not enabled.	
Note	This API can only be called when force sleep function is enabled, after calling wifi_fpm_open. This API can not be called after calling wifi_fpm_close.	
	fpm_wakeup_cb_func will be called after system wakes up only if the force sleep time out (wifi_fpm_do_sleep and the parameter is not 0xFFFFFFF).	
	fpm_wakeup_cb_func will not be called if wake-up is caused by wifi_fpm_do_wakeup from MODEM_SLEEP_T type force sleep.	

3.7.6. wifi_fpm_set_sleep_type

Function	Set sleep type for force sleep function.
Prototype	<pre>void wifi_fpm_set_sleep_type (enum sleep_type type)</pre>



Parameter	<pre>enum sleep_type{ NONE_SLEEP_T = 0, LIGHT_SLEEP_T, MODEM_SLEEP_T, };</pre>	
Return	none	
Note	This API can only be called before wifi_fpm_open.	

3.7.7. wifi_fpm_get_sleep_type

Function	Get sleep type of force sleep function.	
Prototype	enum sleep_type wifi_fpm_get_sleep_type (void)	
Parameter	none	
Return	<pre>enum sleep_type{ NONE_SLEEP_T = 0, LIGHT_SLEEP_T, MODEM_SLEEP_T, };</pre>	

3.7.8. wifi_fpm_auto_sleep_set_in_null_mode

Function	Set whether enter modem sleep mode automatically or not after disabled Wi-Fi mode (wifi_set_opmode(NULL_MODE)).	
Prototype	void wifi_fpm_auto_sleep_set_in_null_mode (uint8 req)	
	uint8 req:	
Parameter	0: disable auto-sleep function;	
	1: enable auto modem sleep when Wi-Fi mode is NULL_MODE.	
Return	none	

3.7.9. Example

For example, forced sleep interface can be called, the RF circuit can be closed mandatorily so as to lower the power.

Note:

When forced sleep interface is called, the chip will not enter sleep mode instantly, it will enter sleep mode when the system is executing idle task. Please refer to the below sample code.

Example one: Modem-sleep mode (disabling RF)

#define FPM_SLEEP_MAX_TIME	0xffffff



```
void fpm_wakup_cb_func1(void)
{
                                               // disable force sleep function
   wifi_fpm_close();
                                               // set station mode
   wifi_set_opmode(STATION_MODE);
   wifi_station_connect();
                                                // connect to AP
void user_func(...)
   wifi_station_disconnect();
   wifi_set_opmode(NULL_MODE);
                                               // set WiFi mode to null mode.
   wifi_fpm_set_sleep_type(MODEM_SLEEP_T);
                                               // modem sleep
   wifi_fpm_open();
                                                // enable force sleep
#ifdef SLEEP_MAX
 /* For modem sleep, FPM_SLEEP_MAX_TIME can only be wakened by calling
wifi_fpm_do_wakeup. */
   wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
#else
   // wakeup automatically when timeout.
   wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1); // Set wakeup callback
   wifi_fpm_do_sleep(50*1000);
#endif
}
 #ifdef SLEEP_MAX
void func1(void)
    wifi_fpm_do_wakeup();
    wifi_fpm_close();
                                              // disable force sleep function
    wifi_set_opmode(STATION_MODE);
                                             // set station mode
                                              // connect to AP
    wifi_station_connect();
}
#endif
```

Example two: Light-sleep mode (disabling RF and CPU)

```
#define FPM_SLEEP_MAX_TIME
                               0xFFFFFF
void fpm_wakup_cb_func1(void)
                                       // disable force sleep function
   wifi_fpm_close();
                                       // set station mode
  wifi_set_opmode(STATION_MODE);
  wifi_station_connect();
                                       // connect to AP
}
#ifndef SLEEP_MAX
// Wakeup till time out.
void user_func(...)
  wifi_station_disconnect();
  wifi_set_opmode(NULL_MODE);
                                       // set WiFi mode to null mode.
   wifi_fpm_set_sleep_type(LIGHT_SLEEP_T);  // light sleep
                                       // enable force sleep
   wifi_fpm_open();
   wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1); // Set wakeup callback
   wifi_fpm_do_sleep(50*1000);
}
#else
// Or wake up by GPIO
void user_func(...)
   wifi_station_disconnect();
   wifi_set_opmode(NULL_MODE);
                                               // set WiFi mode to null mode.
  wifi_fpm_set_sleep_type(LIGHT_SLEEP_T); // light_sleep
   wifi_fpm_open();
                                       // enable force sleep
```



```
PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPI012);
wifi_enable_gpio_wakeup(12, GPI0_PIN_INTR_LOLEVEL);
wifi_fpm_set_wakeup_cb(fpm_wakup_cb_func1); // Set wakeup callback
wifi_fpm_do_sleep(FPM_SLEEP_MAX_TIME);
...
}
#endif
```



3.8. ESP-NOW APIs

ESP-NOW APIs can be found in: /ESP8266_NONOS_SDK/include/espnow.h.

More details about ESP-NOW are in <u>ESP-NOW User Guide</u>. Please note the following points carefully:

- ESP-NOW does not support multicast.
- For *ESP8266_NONOS_SDK_V2.1* and later versions, ESP-NOW supports sending broadcast packets; however, please make sure that the packets are unencrypted.
- It is suggested that slave and combo roles corresponding to SoftAP or SoftAP + Station mode, controller role corresponding to Station mode.
- When ESP8266 is in SoftAP + Station mode, it will communicate through SoftAP interface if it is in slave or combo role, and communicate through Station interface if it is in controller role.
- ESP-NOW can not wake ESP8266 up from sleep, so if the target ESP8266 Station is in sleep, ESP-NOW communication will fail.
- In station mode, ESP8266 supports 10 encrypt ESP-NOW peers at most, with the unencrypted peers, it can be 20 peers in total at most.
- In the SoftAP mode or SoftAP + Station mode, the ESP8266 supports 6 encrypt ESP-NOW peers at most, with the unencrypted peers, it can be 20 peers in total at most.

3.8.1. Roles of ESP-NOW

```
enum esp_now_role {
    ESP_NOW_ROLE_IDLE = 0,
    ESP_NOW_ROLE_CONTROLLER,
    ESP_NOW_ROLE_SLAVE,
    ESP_NOW_ROLE_COMBO,  // both slave and controller
    ESP_NOW_ROLE_MAX,
};
```

3.8.2. esp_now_init

Function	ESP-NOW initialization.
Prototype	init esp_now_init(void)
Parameter	none
Return	0: Success
	otherwise: Failure



3.8.3. esp_now_deinit

Function	Deinitialize ESP-NOW.
Prototype	int esp_now_deinit(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.8.4. esp_now_register_recv_cb

Function	Register ESP-NOW receive callback.
	When received an ESP-NOW packet, enter receive callback:
	<pre>typedef void (*esp_now_recv_cb_t)(u8 *mac_addr, u8 *data, u8 len)</pre>
	Parameters of ESP-NOW receive callback:
	u8 *mac_addr: MAC address of the sender
	u8 *data: data received
	u8 len: data length
Prototype	<pre>int esp_now_register_recv_cb(esp_now_recv_cb_t cb)</pre>
Parameter	esp_now_recv_cb_t cb: receive callback.
Return	0: Success
	otherwise: Failure

3.8.5. esp_now_unregister_recv_cb

Function	Unregister ESP-NOW receive callback.
Prototype	int esp_now_unregister_recv_cb(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.8.6. esp_now_register_send_cb

Function	Register ESP-NOW send callback.
Prototype	u8 esp_now_register_send_cb(esp_now_send_cb_t cb)
Parameter	esp_now_send_cb_t cb: callback.

Espressif 77/166 2018.08



Return	0: Success
neturn	otherwise: Failure
	ESP-NOW send callback:
	<pre>void esp_now_send_cb_t(u8 *mac_addr, u8 status)</pre>
	Parameter:
	u8 *mac_addr: MAC address of target device
	u8 status: status of ESP-NOW sending packet
	mt_tx_status {
	MT_TX_STATUS_OK = 0,
	MT_TX_STATUS_FAILED,
	}
	The status will be updated to MT_TX_STATUS_OK, if ESP-NOW send the packet successfully. Users must explicitly make sure that the key for communication is correct.
	The send callback can determine the status of a transmitted packet, however, please note the following points:
	For unicast packet:
Note	 If the status is OK in the esp_now_send_cb_t, but the target device failed to receive the packet, the reasons may be as follows:
	It may have been corrupted by other unexpected devices;
	2. Incorrect key for communication;
	3. The application was lost or missed the packet.
	Solution: handshake mechanism should be used in application to get a high success rate with packet transmission.
	 If the status is FAIL in the esp_now_send_cb_t, but the target device does receive the packet, the reason may be that the sender fails to receive the ACK because of busy channel.
	Solution: the sender application should re-transmit the packet, and the receiver should detect the retransmitted packet.
	For multicast packet (include broadcast packet):
	 If the status is OK in the esp_now_send_cb_t, it indicates that the packet is sent successfully.
	- If the status is FAIL in the esp_now_send_cb_t , it indicates that sending failed.



3.8.7. esp_now_unregister_send_cb

Function	Unregister ESP-NOW send callback.
Prototype	int esp_now_unregister_send_cb(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.8.8. esp_now_send

Function	Send ESP-NOW packets.
Prototype	int esp_now_send(u8 *da, u8 *data, int len)
Parameter	u8 *da: Destination MAC address. If it's NULL, the packet is sent to all MAC addresses recorded by ESP-NOW; otherwise, the packet is sent to the target MAC address only.
	u8 *data: data to be sent. int len: data length.
	the ten. data longin.
Return	0: Success
	otherwise: Failure

3.8.9. esp_now_add_peer

Function	Add an ESP-NOW peer, store MAC address of target device into ESP-NOW MAC list.
Prototype	<pre>int esp_now_add_peer(u8 *mac_addr, u8 role, u8 channel, u8 *key, u8 key_len)</pre>
Parameter	u8 *mac_addr: MAC address of device.
	u8 role: role type of device; see esp_now_role for details.
	u8 channel: channel of device.
	u8 *key: 16 bytes key which is needed for ESP-NOW communication.
	u8 key_len: length of key, has to be 16 bytes now.
Return	0: Success
	otherwise: Failure

Espressif 79/166 2018.08



3.8.10. esp_now_del_peer

Function	Delete an ESP-NOW peer, delete MAC address of the device from ESP-NOW MAC list.
Prototype	int esp_now_del_peer(u8 *mac_addr)
Parameter	u8 *mac_addr: MAC address of device.
Return	0: Success
	otherwise: Failure

3.8.11. esp_now_set_self_role

Function	Set the ESP-NOW role of the device.
Prototype	int esp_now_set_self_role(u8 role)
Parameter	u8 role: role type; see esp_now_role for details.
Return	0: Success
	otherwise: Failure

3.8.12. esp_now_get_self_role

Function	Get the ESP-NOW role of the device.
Prototype	u8 esp_now_get_self_role(void)
Parameter	none
Return	Role type.

3.8.13. esp_now_set_peer_role

Function	Set the ESP-NOW role for a target device. If it is set more than one times, the new role setting will override the old role.
Prototype	<pre>int esp_now_set_peer_role(u8 *mac_addr,u8 role)</pre>
Parameter	u8 *mac_addr: MAC address of the target device.
	u8 role: role type; see esp_now_role for details.
Return	0: Success
	otherwise: Failure



3.8.14. esp_now_get_peer_role

Function	Get the ESP-NOW role of a target device.
Prototype	int esp_now_get_peer_role(u8 *mac_addr)
Parameter	u8 *mac_addr: MAC address of target device.
Return	esp_now_role: the role type otherwise: Failure

3.8.15. esp_now_set_peer_key

Function	Set ESP-NOW key for a target device. If it is set multiple times, the latest setting will be valid.
Prototype	<pre>int esp_now_set_peer_key(u8 *mac_addr,u8 *key,u8 key_len)</pre>
Parameter	u8 *mac_addr: MAC address of target device.
	u8 *key: 16 bytes key which is needed for ESP-NOW communication; if it is NULL, set the current key to be none.
	u8 key_len: key length; 16 bytes.
Return	0: Success
	otherwise: Failure

3.8.16. esp_now_get_peer_key

Function	Get ESP-NOW key of a target device.
Prototype	int esp_now_set_peer_key(u8 *mac_addr,u8 *key,u8 *key_len)
	u8 *mac_addr: MAC address of target device.
Parameter	u8 *key: pointer to the key, buffer size has to be 16 bytes at least.
	u8 *key_len: key length.
	0: Success
Return	> 0: Found target device but cannot get key
	< 0: Failure

3.8.17. esp_now_set_peer_channel

	Record channel information of an ESP-NOW device.
	To communicate with a device,
Function	call esp_now_get_peer_channel to get its channel first;
	then call wifi_set_channel to be on the same channel and continue further communication.



Prototype	int esp_now_set_peer_channel(u8 *mac_addr,u8 channel)
Parameter	u8 *mac_addr: MAC address of target device.
	u8 channel: channel, usually to be 1 ~ 13, some area may use channel 14.
Return	0: Success
	otherwise: Failure

3.8.18. esp_now_get_peer_channel

Function	Get channel information of a ESP-NOW device. ESP-NOW communication needs to be at the same channel.
Prototype	int esp_now_get_peer_channel(u8 *mac_addr)
Parameter	u8 *mac_addr: MAC address of the target device.
Return	1 ~ 13 (14 for some areas): Success
	otherwise: Failure

3.8.19. esp_now_is_peer_exist

Function	Check if target device exists or not.
Prototype	int esp_now_is_peer_exist(u8 *mac_addr)
Parameter	u8 *mac_addr: MAC address of the target device.
	0: Device does not exist.
Return	< 0: Error, check for device failed.
	> 0: Device exists.

3.8.20. esp_now_fetch_peer

Function	Get MAC address of ESP-NOW device which is pointed now, and move the pointer to next one in ESP-NOW MAC list or move the pointer to the first one in ESP-NOW MAC list.
Prototype	u8 *esp_now_fetch_peer(bool restart)
Parameter	 bool restart: true: move pointer to the first one in ESP-NOW MAC list; false: move pointer to the next one in ESP-NOW MAC list.
Return	none, no ESP-NOW devices exist;Otherwise, MAC address of ESP-NOW device which is pointed now.
Note	This API can not re-entry. Parameter has to be true when you call it the first time.



3.8.21. esp_now_get_cnt_info

Function	Get the total number of ESP-NOW devices which are associated, and the number count of encrypted devices.
Prototype	<pre>int esp_now_get_cnt_info(u8 *all_cnt, u8 *encryp_cnt)</pre>
Parameter	u8 *all_cnt: total number of ESP-NOW devices which are associated.
	u8 *encryp_cnt: number count of encrypted devices.
Return	0: Success
	otherwise: Failure

3.8.22. esp_now_set_kok

Function	Set the encryption key for the communication key. All ESP-NOW devices share the same encrypt key. If users do not set the encrypt key, ESP-NOW communication key will be encrypted by a default key.
	If this API needs to be called, please call it before <pre>esp_now_add_peer</pre> and <pre>esp_now_set_peer_key</pre> .
Prototype	<pre>int esp_now_set_kok(u8 *key, u8 len)</pre>
Parameter	u8 *key: pointer of encryption key;
	u8 len: key length; 16 bytes.
Return	0: Success
	otherwise: Failure



3.9. Simple Pair APIs

Simple Pair APIs can be found in: /ESP8266_NONOS_SDK/include/simple_pair.h.

3.9.1. Status of Simple Pair

```
typedef enum {
        SP_ST_STA_FINISH = 0, // station finished negotiation
        SP\_ST\_AP\_FINISH = 0, // AP finished negotiation
        SP_ST_AP_RECV_NEG,
                             // AP received a request of negotiation from station
        SP_ST_STA_AP_REFUSE_NEG,
                                       // station received the refusal to negotiate
from AP
        /* definitions below are error codes */
        SP_ST_WAIT_TIMEOUT,
                               // Error: time out
        SP_ST_SEND_ERROR,
                              // Error: error occur when sending data
        SP_ST_KEY_INSTALL_ERR, // Error: error occur during key installation
        SP_ST_KEY_OVERLAP_ERR, // Error: one MAC address uses multiple keys
        SP_ST_OP_ERROR,
                             // Error: operational error
        SP_ST_UNKNOWN_ERROR, // Error: unknown error
       SP_ST_MAX,
} SP_ST_t;
```

3.9.2. register_simple_pair_status_cb

Function	Register a callback of status for status of simple pair.
Prototype	<pre>init register_simple_pair_status_cb(simple_pair_status_cb_t cb)</pre>
Callback Definition	typedef void (*simple_pair_status_cb_t)(u8 *sa, u8 status);
	Parameters:
	u8 *sa: the MAC address of the remote device;
	• u8 status: status of simple pair, refer to SP_ST_t.
Parameter	simple_pair_status_cb_t cb: callback.
Return	0: Success
	otherwise: Failure

3.9.3. unregister_simple_pair_status_cb

Function	Unregister the callback of status of simple pair.
Prototype	void unregister_simple_pair_status_cb(void)
Parameter	none



Return	none
--------	------

3.9.4. simple_pair_init

Function	Simple pair initialization.
Prototype	int simple_pair_init(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.9.5. simple_pair_deinit

Function	Deinitialize simple pair.
Prototype	int simple_pair_deinit(void)
Parameter	none
Return	none

3.9.6. simple_pair_state_reset

Function	Reset the state of simple pair. When simple pair needs to be restarted again, this API can be called to reset the state.
Prototype	int simple_pair_state_reset(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.9.7. simple_pair_ap_enter_announce_mode

Function	The AP peer of simple pair enters announce mode.
Prototype	int simple_pair_ap_enter_announce_mode(void)
Parameter	none
Return	0: Success
	otherwise: Failure



3.9.8. simple_pair_sta_enter_scan_mode

Function	The station peer of simple pair enters scan mode.
Prototype	int simple_pair_ap_enter_announce_mode(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.9.9. simple_pair_sta_start_negotiate

Function	The station peer of simple pair starts to negotiate.
Prototype	int simple_pair_sta_start_negotiate(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.9.10. simple_pair_ap_start_negotiate

Function	The AP peer of simple pair agrees to negotiate.
Prototype	int simple_pair_ap_start_negotiate(void)
Parameter	none
Return	0: Success
	otherwise: Failure

3.9.11. simple_pair_ap_refuse_negotiate

Function	The AP peer of simple pair refuses to negotiate.
Prototype	int simple_pair_ap_refuse_negotiate(void)
Parameter	none
Return	0: Success
	otherwise: Failure



3.9.12. simple_pair_set_peer_ref

Function	Set configuration of the peer which needs to negotiate. Note that this only sets the configuration of the peer, and does not install keys or perform any other relevant operations.
	If the device runs as the station peer, this API needs to be called before <pre>simple_pair_sta_start_negotiate.</pre>
	If the device runs as the AP peer, this API needs to be called before simple_pair_ap_start_negotiate or simple_pair_ap_refuse_negotiate.
Prototype	int simple_pair_set_peer_ref(u8 *peer_mac, u8 *tmp_key, u8 *ex_key)
Parameter	u8 *peer_mac: MAC address of the target peer of negotiation, length: 6 bytes, can NOT be null.
	u8 *tmp_key: a temporary key to encrypt the negotiation, length: 16 bytes, can NOT be null.
	u8 *ex_key: a key for exchange, length: 16 bytes. If it is null, the 0x00000000 will be used as the ex_key by default.
Return	0: Success
	otherwise: Failure

$3.9.13.\ simple_pair_get_peer_ref$

Function	Get the configuration of the negotiation. If passing a null pointer, the corresponding parameter will not be got.
Prototype	int simple_pair_get_peer_ref(u8 *peer_mac, u8 *tmp_key, u8 *ex_key)
Parameter	u8 *peer_mac: MAC address of the target peer of negotiation, length: 6 bytes.
	u8 *tmp_key: the temporary key to encrypt the negotiation, length: 16 bytes.
	u8 *ex_key: a key for exchange, length: 16 bytes. If it is null, the 0x00000000 will be used as the ex_key by default.
Return	0: Success
	otherwise: Failure



3.10. Upgrade (FOTA) APIs

FOTA APIs can be found in: **/ESP8266_NONOS_SDK/include/user_interface.h & upgrade.h.**

3.10.1. system_upgrade_userbin_check

Function	Check user bin.
Prototype	uint8 system_upgrade_userbin_check()
Parameter	none
Return	0x00: UPGRADE_FW_BIN1, i.e. <i>user1.bin</i> .
	0x01: UPGRADE_FW_BIN2, i.e. <i>user2.bin</i> .

3.10.2. system_upgrade_flag_set

Function	Set upgrade status flag.
Prototype	<pre>void system_upgrade_flag_set(uint8 flag)</pre>
Parameter	uint8 flag: #define UPGRADE_FLAG_IDLE
Return	none
Note	If you using system_upgrade_start to upgrade, this API need not be called. If you using spi_flash_write to upgrade firmware yourself, this flag need to be set to UPGRADE_FLAG_FINISH, then call system_upgrade_reboot to reboot to run new firmware.

3.10.3. system_upgrade_flag_check

Function	Get upgrade status flag.
Prototype	uint8 system_upgrade_flag_check()
Parameter	none
Return	#define UPGRADE_FLAG_IDLE 0x00 #define UPGRADE_FLAG_START 0x01 #define UPGRADE_FLAG_FINISH 0x02

Espressif 88/166 2018.08



3.10.4. system_upgrade_start

Function	Configure parameters and start upgrading.
Prototype	bool system_upgrade_start (struct upgrade_server_info *server)
Parameter	struct upgrade_server_info *server: server related parameters.
Return	true: start upgrade.
	false: upgrade cannot be started.

3.10.5. system_upgrade_reboot

Function	Reboot system and use new version.
Prototype	void system_upgrade_reboot (void)
Parameter	none
Return	none



3.11. Sniffer Related APIs

Sniffer APIs can be found in: /ESP8266_NONOS_SDK/include/user_interface.h.

3.11.1. wifi_promiscuous_enable

Function	Enable promiscuous mode for sniffer.
Prototype	void wifi_promiscuous_enable(uint8 promiscuous)
	uint8 promiscuous:
Parameter	0: disable promiscuous;
	1: enable promiscuous
Return	none
Note	Promiscuous mode can only be enabled in Station mode.
	During promiscuous mode (sniffer), ESP8266 Station and SoftAP are disabled.
	Before enable promiscuous mode, please call wifi_station_disconnect first.
	Don't call any other APIs during sniffer, please call wifi_promiscuous_enable(0) first.

3.11.2. wifi_promiscuous_set_mac

Function	Set a destination MAC address filter for sniffer. It will filter all packets sent to the specific MAC address, including broadcast packets.
Prototype	void wifi_promiscuous_set_mac(const uint8_t *address)
Parameter	const uint8_t *address: MAC address
Return	none
Note	 This API should be called after calling wifi_promiscuous_enable(1). This filter only be available in the current sniffer phase, if you disable sniffer and then enable sniffer, you need to set filter again if you need it.
Example	<pre>char ap_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab}; wifi_promiscuous_set_mac(ap_mac);</pre>

3.11.3. wifi_set_promiscuous_rx_cb

Function	Registers an Rx callback function in promiscuous mode, which will be called when data packet is received.
Prototype	void wifi_set_promiscuous_rx_cb(wifi_promiscuous_cb_t cb)
Parameter	wifi_promiscuous_cb_t cb: callback
Return	none

Espressif 90/166 2018.08



3.11.4. wifi_get_channel

Function	Get Wi-Fi channel.
Prototype	uint8 wifi_get_channel(void)
Parameter	none
Return	Channel number.

3.11.5. wifi_set_channel

Function	Set Wi-Fi channel, for sniffer mode.
Prototype	bool wifi_set_channel (uint8 channel)
Parameter	uint8 channel: channel number
Return	true: Success
	false: Failure



3.12. Smart Config APIs

SmartConfig APIs can be found in /ESP8266_NONOS_SDK/include/smartconfig.h.

AirKiss APIs can be found in /ESP8266_NONOS_SDK/include/airkiss.h.

Please make sure the target AP is enabled before enabling SmartConfig.

3.12.1. smartconfig_start

Function	Start smart configuration mode. Connect ESP8266 Station to AP, by sniffing for special packets from the air, containing SSID and password of desired AP. You need to broadcast the SSID and password (e.g. from mobile device or computer) with the SSID and password encoded.
Structure	<pre>typedef enum { SC_STATUS_WAIT = 0,</pre>
Prototype	<pre>bool smartconfig_start(sc_callback_t cb, uint8 log)</pre>
Parameter	 sc_callback_t cb: smart config callback; executed when smart-config status changed. Parameter status of this callback shows the status of smart-config: If status == SC_STATUS_GETTING_SSID_PSWD, parameter void *pdata is a pointer of sc_type, means smart-config type: AirKiss or ESP-TOUCH. If status == SC_STATUS_LINK, parameter void *pdata is a pointer of struct station_config; If status == SC_STATUS_LINK_OVER, parameter void *pdata is a pointer of mobile phone's IP address, 4 bytes. This is only available in ESPTOUCH, otherwise, it is NULL. otherwise, parameter void *pdata is NULL. uint8 log: 1: UART outputs logs; otherwise: UART only outputs the result. It is suggested that this log is only used for debugging. Users should not set it to 1 while SmartConfig is working.



```
true: Success
 Return
                 false: Failure
                 void ICACHE_FLASH_ATTR
                 smartconfig_done(sc_status status, void *pdata)
                 {
                     switch(status) {
                         case SC_STATUS_WAIT:
                             os_printf("SC_STATUS_WAIT\n");
                             break;
                         case SC_STATUS_FIND_CHANNEL:
                             os_printf("SC_STATUS_FIND_CHANNEL\n");
                             break;
                         case SC_STATUS_GETTING_SSID_PSWD:
                             os_printf("SC_STATUS_GETTING_SSID_PSWD\n");
                             sc_type *type = pdata;
                             if (*type == SC_TYPE_ESPTOUCH) {
                                  os_printf("SC_TYPE:SC_TYPE_ESPTOUCH\n");
                             } else {
                                  os_printf("SC_TYPE:SC_TYPE_AIRKISS\n");
                             }
                             break;
Example
                         case SC_STATUS_LINK:
                             os_printf("SC_STATUS_LINK\n");
                             struct station_config *sta_conf = pdata;
                             wifi_station_set_config(sta_conf);
                             wifi_station_disconnect();
                                  wifi_station_connect();
                             break;
                         case SC_STATUS_LINK_OVER:
                             os_printf("SC_STATUS_LINK_OVER\n");
                                  if (pdata != NULL) {
                                  uint8 phone_ip[4] = \{0\};
                                  memcpy(phone_ip, (uint8*)pdata, 4);
                                  os_printf("Phone ip: %d.%d.%d.
                 %d\n",phone_ip[0],phone_ip[1],phone_ip[2],phone_ip[3]);
                             smartconfig_stop();
                             break;
                     }
                 }
                 smartconfig_start(smartconfig_done);
                 • This API can only be called in station mode.
                   During smart-config, ESP8266 Station and SoftAP are disabled.
                 • Can not call smartconfig_start twice before it finish, please call
 Note
                    smartconfig_stop first.
                 • Don't call any other APIs during smart-config, please call smartconfig_stop
                    first.
```



3.12.2. smartconfig_stop

Function	Stop smart config, free the buffer taken by smartconfig_start.
Prototype	bool smartconfig_stop(void)
Parameter	none
Return	true: Success
	false: Failure
Note	Irrespective of whether connection to AP succeeded or not, this API should be called to free memory taken by smartconfig_start.

3.12.3. smartconfig_set_type

Function	Set the protocol type of SmartConfig.
Prototype	bool smartconfig_set_type(sc_type type)
Parameter	<pre>typedef enum { SC_TYPE_ESPTOUCH = 0, SC_TYPE_AIRKISS, SC_TYPE_ESPTOUCH_AIRKISS, } sc_type;</pre>
Return	true: Success false: Failure
Note	This API can only be called before calling smartconfig_start.

3.12.4. airkiss_version

Function	Get version information of the AirKiss lib.
Prototype	const char* airkiss_version(void)
Parameter	none
Return	Version information of the AirKiss lib.
Note	The length of the version information is unknown.



3.12.5. airkiss_lan_recv

	For the function that AirKiss can detect the ESP8266 devices in LAN, more details of this function refer to http://iot.weixin.qq.com .
Function	Workflow: Create a UDP transmission. When UDP data is received in espconn_recv_callback, call API airkiss_lan_recv and input the UDP data; if airkiss_lan_recv returns AIRKISS_LAN_SSDP_REQ, airkiss_lan_pack can be called to make a response packet.
	This API is to parse the UDP packet sent by WeChat.
Prototype	<pre>int airkiss_lan_recv(const void* body, unsigned short length, const airkiss_config_t* config)</pre>
Parameter	<pre>const void* body: the received UDP packet; unsigned short length: the length of UDP packet; airkiss_config_t* config: AirKiss structure.</pre>
Return	Refer to airkiss_lan_ret_t, >= 0: Success, < 0: Failure

3.12.6. airkiss_lan_pack

Function	User packet assembly for the function that AirKiss can detect the ESP8266 devices in LAN.
Prototype	<pre>int airkiss_lan_pack(</pre>
Parameter	airkiss_lan_cmdid_t ak_lan_cmdid: packet type. void* appid: WeChat public number, got from WeChat. void* deviceid: device ID, got from WeChat. void* _datain: user data waiting for packet assembly. unsigned short inlength: length of the user data. void* _dataout: the packet got by _datain packet assembly. unsigned short* outlength: length of the packet. const airkiss_config_t* config: AirKiss structure.
Return	Refer to airkiss_lan_ret_t >= 0: Success, < 0: Failure



3.13. SNTP APIs

SNTP APIs can be found in: /ESP8266_NONOS_SDK/include/sntp.h.

3.13.1. sntp_setserver

Function	Set SNTP server by IP address; it supports 3 SNTP servers at most.
Prototype	void sntp_setserver(unsigned char idx, ip_addr_t *addr)
Parameter	unsigned char idx: SNTP server index; 3 SNTP server is supported at most (0 ~ 2); index 0 is the main server, and index 1 and 2 are backups.
	ip_addr_t *addr: IP address; users need to ensure that it is an SNTP server.
Return	none

3.13.2. sntp_getserver

Function	Get IP address of SNTP server as set by sntp_setserver.
Prototype	ip_addr_t sntp_getserver(unsigned char idx)
Parameter	unsigned char idx: SNTP server index; supports 3 SNTP servers at most (0 ~ 2).
Return	IP address

3.13.3. sntp_setservername

Function	Set SNTP server by domain name; support 3 SNTP server at most.
Prototype	void sntp_setservername(unsigned char idx, char *server)
Parameter	unsigned char idx: SNTP server index, supports 3 SNTP servers at most (0 ~ 2); index 0 is the main server, and index 1 and 2 are as backup.
	char *server: domain name; users need to ensure that it is an SNTP server.
Return	none

3.13.4. sntp_getservername

Function	Get domain name of SNTP server which set by sntp_setservername.
Prototype	char * sntp_getservername(unsigned char idx)
Parameter	unsigned char idx: SNTP server index; supports 3 SNTP servers at most (0 ~ 2).
Return	Domain name

Espressif 96/166 2018.08



3.13.5. sntp_init

Function	Initialize SNTP.
Prototype	void sntp_init(void)
Parameter	none
Return	none

3.13.6. sntp_stop

Function	Stop SNTP.
Prototype	void sntp_stop(void)
Parameter	none
Return	none

3.13.7. sntp_get_current_timestamp

Function	Get current timestamp from basic time (1970.01.01 00:00:00 GMT + 8); uint: second.
Prototype	uint32 sntp_get_current_timestamp()
Parameter	none
Return	Time stamp

3.13.8. sntp_get_real_time

Function	Get real time(GMT + 8)
Prototype	char* sntp_get_real_time(long t)
Parameter	long t: time stamp
Return	Real time

3.13.9. sntp_set_timezone

Function	Set time zone.
Prototype	bool sntp_set_timezone (sint8 timezone)
Parameter	sint8 timezone: time zone; range:-11 ~ 13.
Return	true: Success
	false: Failure



3.13.10.sntp_get_timezone

Function	Get time zone.
Prototype	sint8 sntp_get_timezone (void)
Parameter	none
Return	Time zone; range: -11 ~ 13.

3.13.11.SNTP Example

Step 1. Enable SNTP.

```
ip_addr_t *addr = (ip_addr_t *)os_zalloc(sizeof(ip_addr_t));
sntp_setservername(0, "us.pool.ntp.org"); // set server 0 by domain name
sntp_setservername(1, "ntp.sjtu.edu.cn"); // set server 1 by domain name
ipaddr_aton("210.72.145.44", addr);
sntp_setserver(2, addr); // set server 2 by IP address
sntp_init();
os_free(addr);
```

Step 2. Set a timer to check SNTP timestamp.

```
LOCAL os_timer_t sntp_timer;

os_timer_disarm(&sntp_timer);

os_timer_setfn(&sntp_timer, (os_timer_func_t *)user_check_sntp_stamp, NULL);

os_timer_arm(&sntp_timer, 100, 0);
```

Step 3. Timer Callback

```
void ICACHE_FLASH_ATTR user_check_sntp_stamp(void *arg){
    uint32 current_stamp;
    current_stamp = sntp_get_current_timestamp();
    if(current_stamp == 0){
```





3.14. WPA2_Enterprise APIs

ESP8266 Station can connect to WPA2_Enterprise APs.

WPA2_Enterprise APIs can be found in /ESP8266_NONOS_SDK/include/wpa2_enterprise.h.

3.14.1. wifi_station_set_wpa2_enterprise_auth

	Set authentication of WPA2_Enterprise.
Function	To connect to WPA2_Enterprise AP, wifi_station_set_wpa2_enterprise_auth(1); should be called first. For connecting to a regular AP at a later stage, wifi_station_set_wpa2_enterprise_auth(0); should be called to clear the WPA2_Enterprise status.
Prototype	int wifi_station_set_wpa2_enterprise_auth(int enable)
Parameter	int enable:
	0, disable authentication of WPA2_Enterprise, clear the status;
	otherwise, enable authentication of WPA2_Enterprise.
Return	0: Success
	otherwise: Failure

3.14.2. wifi_station_set_enterprise_cert_key

Function	Set user certificate and private key for connecting to WPA2_Enterprise AP. It is used for EAP-TLS authentication.
Prototype	<pre>int wifi_station_set_enterprise_cert_key (u8 *client_cert, int client_cert_len, u8 *private_key, int private_key_len, u8 *private_key_passwd, int private_key_passwd_len,)</pre>
	u8 *client_cert: user certificate, HEX array
	int client_cert_len: length of certificate
	u8 *private_key: private key, HEX array, can NOT be longer than 2048 bits
Parameter	int private_key_len: length of private key, less than 2048
	<pre>u8 *private_key_passwd: password for private key, to be supported, can only be NULL now.</pre>
	<pre>int private_key_passwd_len: length of password, to be supported, can only be 0 now.</pre>
Return	0: Success
	otherwise: Failure

Espressif 100/166 2018.08



	For example, the private key is BEGIN PRIVATE KEY
Example	then array should be uint8 key[]={0x2d, 0x2d, 0x2d, 0x2d, 0x2d, 0x42, 0x45, 0x47, 0x00 };
	It is the ASCII code for the characters, and the array needs to terminate with 0x00.
	Connecting to WPA2-ENTERPRISE AP needs more than 26 KB memory, please ensure enough space (system_get_free_heap_size).
	So far, WPA2-ENTERPRISE can only support unencrypted certificate and private key, and only in PEM format.
	- Header of certificate: BEGIN CERTIFICATE
	- Header of private key: BEGIN RSA PRIVATE KEY or BEGIN PRIVATE KEY
	Please call this API to set certificate and private key before connecting to WPA2_ Enterprise AP and the application needs to hold the certificate and private key. Call wifi_station_clear_enterprise_cert_key to release resources and clear status after being connected to the target AP, and then the application can release the certificate and private key.
	If the private key is encrypted, please use openssl pkey command to change it to unencrypted file to use, or use openssl rsa related commands to change it (or change the start TAG).

3.14.3. wifi_station_clear_enterprise_cert_key

Function	Release user certificate and private key resources and clear related status after being connected to the WPA2_Enterprise AP.
Prototype	void wifi_station_clear_enterprise_cert_key (void)
Parameter	none
Return	none

${\bf 3.14.4.\ wifi_station_set_enterprise_ca_cert}$

Function	Set root certificate for connecting to WPA2_Enterprise AP. It is an option in EAP-TTLS/PEAP authentication.
Prototype	int wifi_station_set_enterprise_ca_cert(u8 *ca_cert, int ca_cert_len)
Parameter	u8 *ca_cert: root certificate, HEX array
	int ca_cert_len: length of root certificate
Return	0 : Success
	otherwise : Failure

Espressif 101/166 2018.08



3.14.5. wifi_station_clear_enterprise_ca_cert

Function	Release root certificate resources and clear related status after being connected to the WPA2_Enterprise AP.
Prototype	void wifi_station_clear_enterprise_ca_cert (void)
Parameter	none
Return	none

3.14.6. wifi_station_set_enterprise_username

Function	Set ESP8266 Station's user name for connecting to WPA2_Enterprise AP.
Prototype	int wifi_station_set_enterprise_username (u8 *username, int len)
Parameter	u8 *username: the user name
	int len: length of user name
Return	0: Success
	otherwise: Failure
Note	For EAP-TTLS and EAP-PEAP authentication, the user name has to be set. It is used in phase 2 of the authentication, only the user name that the server supported can pass the authentication.
	For EAP-TTLS and EAP-PEAP authentication, the user name is only an option. Without setting user name, the authentication can still be done anonymously.

3.14.7. wifi_station_clear_enterprise_username

Function	Release the user name resources and clear related status after being connected to the WPA2_Enterprise AP.
Prototype	void wifi_station_clear_enterprise_username (void)
Parameter	none
Return	none

3.14.8. wifi_station_set_enterprise_password

Function	Set the password for connecting to WPA2_Enterprise AP. It is used for EAP-TTLS / EAP-PEAP authentication.
Prototype	int wifi_station_set_enterprise_password (u8 *password, int len)
Parameter	u8 *password: the user password int len: length of the password
Return	0: Success otherwise: Failure

Espressif 102/166 2018.08



3.14.9. wifi_station_clear_enterprise_password

Function	Clear the password resources and clear related status data after being connected to the WPA2_Enterprise AP.
Prototype	void wifi_station_clear_enterprise_password (void)
Parameter	none
Return	none

3.14.10.wifi_station_set_enterprise_new_password

Function	Set the new password for connecting to WPA2_Enterprise AP. It is used for MSCHAPV2.
Prototype	<pre>int wifi_station_set_enterprise_new_password (u8 *new_password, int len)</pre>
Parameter	u8 *new_password: the new password int len: length of the password
Return	0: Success otherwise: Failure

3.14.11.wifi_station_clear_enterprise_new_password

Function	Release the new password resources and clear related status after being connected to the WPA2_Enterprise AP.
Prototype	void wifi_station_clear_enterprise_new_password (void)
Parameter	none
Return	none

3.14.12.wifi_station_set_enterprise_disable_time_check

Function	Determines whether expiration time is checked in authentication. The expiration time will not be checked by default.
Prototype	void wifi_station_set_enterprise_disable_time_check (bool disable)
Parameter	 bool disable: true, will NOT check the expiration time; false, check the expiration time, wpa2_enterprise_set_user_get_time has to be called.
Return	none

Espressif 103/166 2018.08



3.14.13.wifi_station_get_enterprise_disable_time_check

Function	Check whether the expiration time will be observed in authentication.
Prototype	bool wifi_station_get_enterprise_disable_time_check (void)
Parameter	none
Return	True: will NOT check the expiration time
	False: check the expiration time

3.14.14.wpa2_enterprise_set_user_get_time

Function	Set time callback to get current time from user. wifi_station_set_enterprise_disable_time_check(false); should be called as the example below.
Prototype	void wpa2_enterprise_set_user_get_time(get_time_func_t cb)
Parameter	get_time_func_t cb: callback
Return	none
Example	<pre>static int sys_get_current_time(struct os_time *t) {</pre>

3.14.15.WPA2_Enterprise Work Flow

Here is the work flow that prepares ESP266 station to connect to WPA2_Enterprise AP.

- Call wifi_station_set_config to set the configuration of target AP.
- Call wifi_station_set_wpa2_enterprise_auth(1); to enable WPA2_Enterprise authentication.
 - For EAP-TLS authentication, call wifi_station_set_enterprise_cert_key to set certificate and private key. wifi_station_set_enterprise_username is an optional choice, it can be called to set user name.
 - For EAP-TTLS or EAP-PEAP authentication, call
 wifi_station_set_enterprise_username and
 wifi_station_set_enterprise_password to set user name and password.
 wifi_station_set_enterprise_ca_cert is an optional choice, it can be called to set root certificate.
- Call wifi_station_connect to connect to target AP.

Espressif 104/166 2018.08



• After being connected to an AP, or failing to connect to AP and on stopped retries, please call the corresponding wifi_station_clear_enterprise_XXX APIs to release the resources.



4.

TCP/UDP APIs

Found in *ESP8266_NONOS_SDK/include/espconn.h*. The network APIs can be grouped into the following types:

- General APIs: APIs can be used for both TCP and UDP.
- TCP APIs: APIs that are only used for TCP.
- UDP APIs: APIs that are only used for UDP.
- mDNS APIs: APIs that related to mDNS.

4.1. Generic TCP/UDP APIs

4.1.1. espconn_delete

Function	Delete a transmission.
Prototype	sint8 espconn_delete(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding connected control block structure
	0: Success
	Otherwise: error, return error code
Return	ESPCONN_ARG: illegal argument, cannot find network transmission according to structure espconn.
	ESPCONN_INPROGRESS: the connection is still in progress; please call espconn_disconnect to disconnect before deleting it.
	Corresponding API:
Note	TCP: espconn_accept;
	UDP: espconn_create.

4.1.2. espconn_gethostbyname

Function	DNS
Prototype	err_t espconn_gethostbyname(struct espconn *pespconn, const char *hostname, ip_addr_t *addr, dns_found_callback found)

Espressif 106/166 2018.08



	struct espconn *espconn: corresponding connected control block structure
Parameter	const char *hostname: domain name string pointer
	ip_addr_t *addr: IP address
	dns_found_callback found: callback
	err_t:
	ESPCONN_OK: Success.
Return	ESPCONN_INPROGRESS: Error code : already connected.
	ESPCONN_ARG: Error code: illegal argument; cannot find network transmission according to structure espconn.
	Corresponding creation API:
Note	TCP: espconn_accept;
	UDP: espconn_create.
	<pre>ip_addr_t esp_server_ip; LOCAL void ICACHE_FLASH_ATTR user_esp_platform_dns_found(const char *name, ip_addr_t *ipaddr, void *arg) { struct espconn *pespconn = (struct espconn *)arg;</pre>
Example	<pre>if (ipaddr != NULL) os_printf(user_esp_platform_dns_found %d.%d.%d.%d/n, *((uint8 *)&ipaddr->addr), *((uint8 *)&ipaddr->addr + 1), *((uint8 *)&ipaddr->addr + 2), *((uint8 *)&ipaddr->addr + 3)); } void dns_test(void) { espconn_gethostbyname(pespconn,"iot.espressif.cn", &esp_server_ip,</pre>

4.1.3. espconn_port

Function	Get an available port.
Prototype	uint32 espconn_port(void)
Parameter	none
Return	uint32: ID of the port you get

4.1.4. espconn_regist_sentcb

Function	Register data sent function which will be called back when data are successfully sent.
Prototype	<pre>sint8 espconn_regist_sentcb(struct espconn *espconn, espconn_sent_callback sent_cb)</pre>



Parameter	struct espconn *espconn: corresponding connected control block structure
	espconn_sent_callback sent_cb: registered callback function
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find network transmission according to structure espconn.

4.1.5. espconn_regist_recvcb

Function	Register data receive function which will be called back when data is received.
Prototype	<pre>sint8 espconn_regist_recvcb(struct espconn *espconn, espconn_recv_callback recv_cb)</pre>
Parameter	struct espconn *espconn: corresponding connected control block structure; espconn_connect_callback connect_cb: registered callback function.
Return	0: Success Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find network transmission according to structure espconn.

4.1.6. espconn_sent_callback

Function	Callback after the data are sent.
Prototype	void espconn_sent_callback (void *arg)
Parameter	void *arg: pointer corresponding structure espconn. This pointer may be different in different callbacks, please don't use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.
Return	none

4.1.7. espconn_recv_callback

Function	Callback after data are received.
Prototype	<pre>void espconn_recv_callback (void *arg, char *pdata, unsigned short len)</pre>



Parameter	void *arg: pointer corresponding structure espconn. This pointer may be different in different callbacks, please don't use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead.
	char *pdata: received data entry parameters.
	unsigned short len: received data length.
Return	none

4.1.8. espconn_get_connection_info

Function	Get the information about a TCP connection or UDP transmission. Usually used in the espconn_recv_callback.
Prototype	<pre>sint8 espconn_get_connection_info(struct espconn *espconn, remot_info **pcon_info, uint8 typeflags)</pre>
Parameter	 struct espconn *espconn: corresponding connected control block structure; remot_info **pcon_info: connect to client info; uint8 typeflags: 0, regular server; 1, ssl server.
Return	0: Success



```
void user_udp_recv_cb(void *arg, char *pusrdata, unsigned short
                length)
                 {
                     struct espconn *pesp_conn = arg;
                     remot_info *premot = NULL;
                     if (espconn_get_connection_info(pesp_conn,&premot,0) ==
                ESPCONN_OK){
                             pesp_conn->proto.tcp->remote_port = premot->remote_port;
                             pesp_conn->proto.tcp->remote_ip[0] = premot-
Example
                >remote_ip[0];
                             pesp_conn->proto.tcp->remote_ip[1] = premot-
                >remote_ip[1];
                             pesp_conn->proto.tcp->remote_ip[2] = premot-
                >remote_ip[2];
                             pesp_conn->proto.tcp->remote_ip[3] = premot-
                >remote_ip[3];
                             espconn_sent(pesp_conn, pusrdata, os_strlen(pusrdata));
                     }
                 }
```

4.1.9. espconn_send

Function	Send data through network.
Prototype	<pre>sint8 espconn_send(struct espconn *espconn, uint8 *psent, uint16 length)</pre>
Parameter	struct espconn *espconn: corresponding connected control block structure; uint8 *psent: pointer of sent data; uint16 length: data length.
Return	O: Success Otherwise: Error code • ESPCONN_MEM: Out of memory; • ESPCONN_ARG: illegal argument; cannot find network transmission according to structure espconn; • ESPCONN_MAXNUM: buffer (or 8 packets at most) of sending data is full; • ESPCONN_IF: fail to send UDP data.
Note	 Please call espconn_send after espconn_sent_callback of the pre-packet. If it is a UDP transmission, please set espconn->proto.udp->remote_ip and remote_port before every calling of espconn_send.



4.1.10. espconn_sent

[@deprecated] This API is deprecated, please use espconn_send instead.

Function	Send data through network.
Prototype	<pre>sint8 espconn_send(struct espconn *espconn, uint8 *psent, uint16 length)</pre>
Parameter	struct espconn *espconn: corresponding connected control block structure; uint8 *psent: pointer of sent data; uint16 length: data length.
Return	O: Success Otherwise: Error code • ESPCONN_MEM: Out of memory; • ESPCONN_ARG: illegal argument; cannot find network transmission according to structure espconn; • ESPCONN_MAXNUM: buffer (or 8 packets at most) of sending data is full; • ESPCONN_IF: fail to send UDP data.
Note	 Please call espconn_send after espconn_sent_callback of the pre-packet. If it is a UDP transmission, please set espconn->proto.udp->remote_ip and remote_port before every calling of espconn_send.

4.2. TCP APIs

TCP APIs act only on TCP connections and do not affect nor apply to UDP connections.

4.2.1. espconn_accept

Function	Creates a TCP server (i.e. accepts connections).
Prototype	sint8 espconn_accept(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding connected control block structure.
Return	0: Success
	Otherwise: Error code
	ESPCONN_MEM: Out of memory;
	ESPCONN_ISCONN: Already connected;
	ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

Espressif 111/166 2018.08



4.2.2. espconn_regist_time

Function	Register timeout interval of ESP8266 TCP server.
Prototype	<pre>sint8 espconn_regist_time(struct espconn *espconn, uint32 interval, uint8 type_flag)</pre>
Parameter	<pre>struct espconn *espconn: corresponding connected control block structure; uint32 interval: timeout interval; unit: second; maximum: 7200 seconds; uint8 type_flag: 0, set all connections; 1, set a single connection.</pre>
Return	0: Success. ESPCONN_ARG: illegal argument, cannot find TCP connection according to structure espconn.
Note	 Call this API after espconn_accept, before listening to a TCP connection. This API can not be used for SSL connection. This timeout interval is not precise. Please use it only as a reference. If timeout is set to 0, timeout will be disabled and ESP8266 TCP server will not disconnect if a TCP client has stopped communication. This usage of timeout=0, is deprecated.

4.2.3. espconn_connect

Function	Connect to a TCP server (ESP8266 acting as TCP client).
Prototype	sint8 espconn_connect(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding connected control block structure.
Return	O: Success Otherwise: Error code • ESPCONN_RTE: Routing Problem • ESPCONN_MEM: Out of memory • ESPCONN_ISCONN: Already connected • ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.
Note	If espconn_connect fails, returns otherwise value. There is no connection, so it won't enter any espconn callback. It is recommended to use espconn_port to get an available local port.

Espressif 112/166 2018.08



4.2.4. espconn_regist_connectcb

Function	Register a connected callback which will be called on successful TCP connection.
Prototype	<pre>sint8 espconn_regist_connectcb(struct espconn *espconn, espconn_connect_callback connect_cb)</pre>
Parameter	struct espconn *espconn: corresponding connected control block structure; espconn_connect_callback connect_cb: registered callback function.
Return	0: Success Otherwise: error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

4.2.5. espconn_connect_callback

Function	Callback for successful connection (ESP8266 as TCP server or ESP8266 as TCP client).
	Callback can be registered by espconn_regist_connectcb.
Prototype	void espconn_connect_callback (void *arg)
Parameter	void *arg: pointer to corresponding structure espconn. This pointer may be different in different callbacks. Please don't use this pointer directly to distinguish one from another in multiple connections; use remote_ip and remote_port in espconn instead.
Return	none

4.2.6. espconn_set_opt

Function	Set configuration options for TCP connection.
Prototype	sint8 espconn_set_opt(struct espconn *espconn, uint8 opt)
Structure	<pre>enum espconn_option{ ESPCONN_START = 0x00, ESPCONN_REUSEADDR = 0x01, ESPCONN_NODELAY = 0x02, ESPCONN_COPY = 0x04, ESPCONN_KEEPALIVE = 0x08, ESPCONN_END }</pre>



Parameter	struct espconn *espconn: corresponding connected control structure.
	uint8 opt: options for TCP connection; please refer to espconn_option.
	 bit 0: 1: free memory after TCP disconnection. Need not wait for 2 minutes.
	 bit 1: 1: disable nagle algorithm during TCP data transmission, thus quickening the data transmission.
	- bit 2: 1: enable espconn_regist_write_finish. Enter write finish callback once the data has been sent using espconn_send (data was written to 2920 bytes write-buffer for sending or has already been sent).
	- bit 3: 1: enable TCP keep alive.
Return	0: Success
	Otherwise: error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

4.2.7. espconn_clear_opt

Function	Clear option of TCP connection.
Prototype	<pre>sint8 espconn_clear_opt(</pre>
	<pre>enum espconn_option{</pre>
	$ESPCONN_START = 0x00,$
	ESPCONN_REUSEADDR = 0x01,
Structure	ESPCONN_NODELAY = 0x02,
Ottuotuic	$ESPCONN_COPY = 0 \times 04,$
	ESPCONN_KEEPALIVE = 0x08,
	ESPCONN_END
	}
Parameter	struct espconn *espconn: corresponding connected control block structure
raiameter	uint8 opt: option of TCP connection, refer to espconn_option.
Return	0: Success
	Otherwise: error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

4.2.8. espconn_set_keepalive

Function	Set configuration of TCP keep alive.
Prototype	<pre>sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void* optarg)</pre>



Structure	enum espconn_level{
	ESPCONN_KEEPIDLE,
	ESPCONN_KEEPINTVL,
	ESPCONN_KEEPCNT
	}
	struct espconn *espconn: corresponding connected control block structure
Parameter	• uint8 level: Default to do TCP keep-alive detection every ESPCONN_KEEPIDLE, if there in no response, retry ESPCONN_KEEPCNT times every ESPCONN_KEEPINTVL. If there is still no response, it is considered as a broken TCP connection and program calls espconn_reconnect_callback.
	Please note that keep alive interval is not precise, only for reference, it depends on priority.
i arameter	Description:
	- ESPCONN_KEEPIDLE: TCP keep-alive interval; unit:second.
	 ESPCONN_KEEPINTVL: packet interval during TCP keep-alive; unit: second.
	- ESPCONN_KEEPCNT: maximum packet count of TCP keep-alive.
	• void* optarg: value of parameter.
	0: Success
Return	Otherwise: error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.
Note	In general, this API need not be called.
	If needed, please call it in espconn_connect_callback and call espconn_set_opt to enable keep alive first.

4.2.9. espconn_get_keepalive

Function	Get value of TCP keep-alive parameter.
Prototype	<pre>sint8 espconn_set_keepalive(struct espconn *espconn, uint8 level, void* optarg)</pre>
Structure	<pre>enum espconn_level{ ESPCONN_KEEPIDLE, ESPCONN_KEEPINTVL, ESPCONN_KEEPCNT }</pre>



Parameter	struct espconn *espconn: corresponding connected control block structure.
	• uint8 level:
	- ESPCONN_KEEPIDLE: TCP keep-alive interval; unit:second.
	 ESPCONN_KEEPINTVL: packet interval during TCP keep-alive; unit: second.
	- ESPCONN_KEEPCNT: maximum packet count of TCP keep-alive.
	void* optarg: value of parameter.
Return	0: Success
	Otherwise: error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

4.2.10. espconn_reconnect_callback

Function	This callback is entered when an error occurs, TCP connection broken. This callback is registered by espconn_regist_reconcb.
Prototype	void espconn_reconnect_callback (void *arg, sint8 err)
Structure	<pre>enum espconn_level{ ESPCONN_KEEPIDLE, ESPCONN_KEEPINTVL, ESPCONN_KEEPCNT }</pre>
Parameter	 void *arg: pointer corresponding structure espconn. This pointer may be different in different callbacks, please do not use this pointer directly to distinguish one from another in multiple connections, use remote_ip and remote_port in espconn instead. sint8 err: error code ESCONN_TIMEOUT: Timeout. ESPCONN_ABRT: TCP connection aborted. ESPCONN_CLSD: TCP connection reset. ESPCONN_CLSD: TCP connection fails. ESPCONN_HANDSHAKE: TCP SSL handshake fails. ESPCONN_PROTO_MSG: SSL application is invalid.
Return	none



4.2.11. espconn_regist_reconcb

Function	Register reconnection callback.
Prototype	<pre>sint8 espconn_regist_reconcb(struct espconn *espconn, espconn_reconnect_callback recon_cb)</pre>
Structure	<pre>enum espconn_level{ ESPCONN_KEEPIDLE, ESPCONN_KEEPINTVL, ESPCONN_KEEPCNT }</pre>
Parameter	 struct espconn *espconn: corresponding connected control block structure; espconn_reconnect_callback recon_cb: registered callback function.
Return	0: Success. Otherwise: Error code ESPCONN_ARG: illegal argument, cannot find TCP connection according to structure espconn.
Note	espconn_reconnect_callback is more like a network-broken error handler; it handles errors that occurs in any phase of the connection. For instance, if espconn_send fails, the network is assumed to have broken and espconn_reconnect_callback.

4.2.12. espconn_disconnect

Function	Disconnect a TCP connection.
Prototype	sint8 espconn_disconnect(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding connected control structure.
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument, cannot find TCP connection according to structure espconn.
Note	Do not call this API in any espconn callback. If needed, please use system_os_task and system_os_post to trigger espconn_disconnect.

4.2.13. espconn_regist_disconcb

Function	Disconnect a TCP connection. Register disconnection function which will be called back under successful TCP disconnection.
Prototype	sint8 espconn_regist_disconcb(struct espconn *espconn, espconn_connect_callback discon_cb)



Parameter	struct espconn *espconn: corresponding connected control block structure.
	espconn_connect_callback connect_cb: registered callback function.
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument, cannot find TCP connection according to structure espconn.

4.2.14. espconn_abort

Function	Forcefully abort a TCP connection.
Prototype	sint8 espconn_abort(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection.
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.
Note	Do not call this API in any espconn callback. If needed, please use system_os_task and system_os_post to trigger espconn_abort.

4.2.15. espconn_regist_write_finish

Function	Register a callback which will be called when all sending data is completely written into write buffer or sent.
Prototype	<pre>sint8 espconn_regist_write_finish (struct espconn *espconn, espconn_connect_callback write_finish_fn)</pre>
Parameter	struct espconn *espconn: corresponding network connection espconn_connect_callback write_finish_fn: registered callback function
Notes	This API can not be used for SSL connection.
	write-buffer is used to keep TCP data that is waiting to be sent, queue number of the write-buffer is 8 which means that it can keep 8 packets at most. The size of write-buffer is 2920 bytes.
	Users can enable it by using espconn_set_opt.
	Users can call espconn_send to send the next packet in write_finish_callback instead of using espconn_sent_callback.
Return	0 : Success Non-0 : Error code ESPCONN_ARG - illegal argument, cannot find TCP connection according to structure espconn

Espressif 118/166 2018.08



4.2.16. espconn_tcp_get_max_con

Function	Get the number of maximum TCP connections allowed.
Prototype	uint8 espconn_tcp_get_max_con(void)
Parameter	none
Return	Maximum number of TCP connections allowed.

4.2.17. espconn_tcp_set_max_con

Function	Set the maximum number of TCP connections allowed.
Prototype	sint8 espconn_tcp_set_max_con(uint8 num)
Parameter	uint8 num: Maximum number of TCP connections allowed.
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

4.2.18. espconn_tcp_get_max_con_allow

Function	Get the maximum number of TCP clients allowed to connect to ESP8266 TCP server.
Prototype	sint8 espconn_tcp_get_max_con_allow(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection.
Return	> 0: Maximum number of TCP clients allowed.
	< 0: Error code ESPCONN_ARG: illegal argument, cannot find TCP connection according to structure espconn.

4.2.19. espconn_tcp_set_max_con_allow

Function	Set the maximum number of TCP clients allowed to connect to ESP8266 TCP server.
Prototype	<pre>sint8 espconn_tcp_set_max_con_allow(struct espconn *espconn, uint8 num)</pre>
Parameter	 struct espconn *espconn: corresponding network connection. uint8 num: Maximum number of TCP clients allowed.
Return	0: Success Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

Espressif 119/166 2018.08



4.2.20. espconn_recv_hold

Function	Puts in a request to block the TCP receive function.
Prototype	sint8 espconn_recv_hold(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection.
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.

4.2.21. espconn_recv_unhold

Function	Unblock TCP receiving data (i.e. undo espconn_recv_hold).
Prototype	sint8 espconn_recv_unhold(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection.
Return	0: Success Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find TCP connection according to structure espconn.
Note	This API takes effect immediately.

4.2.22. espconn_secure_accept

Function	Creates an SSL server.
Prototype	sint8 espconn_secure_accept(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection.
Return	0: Success
	Otherwise: Error code
	ESPCONN_MEM: Out of memory.
	ESPCONN_ISCONN: Already connected.
	ESPCONN_ARG: illegal argument; cannot find connection according to structure espconn.

Espressif 120/166 2018.08



	This API can be called only once. Only one SSL server is allowed to be created, and only one SSL client can be connected.
	If the SSL encrypted packet size is larger than ESP8266's SSL buffer size (default 2 KB, set by espconn_secure_set_size), SSL connection will fail. ESP8266 will enter espconn_reconnect_callback.
Note	• SSL-related APIs named as espconn_secure_xxx are different from normal TCP APIs and must not be used interchangeably. In SSL connection, only espconn_secure_xxx APIs, espconn_regist_xxxxcb APIs and espconn_port can be used.
	• Users should call API espconn_secure_set_default_certificate and espconn_secure_set_default_private_key to set SSL certificate and secure key first.

4.2.23. espconn_secure_delete

Function	Delete the SSL connection when ESP8266 runs as SSL server.
Prototype	sint8 espconn_secure_delete(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding SSL connection.
	0: Success; Otherwise: Error, return error code.
Return	ESPCONN_ARG: illegal argument, cannot find network transmission according
	to structure espconn.
	ESPCONN_INPROGRESS: the SSL connection is still in progress, please call espconn_secure_disconnect to disconnect before deleting it.

4.2.24. espconn_secure_set_size

Function	Set buffer size of encrypted data (SSL).
Prototype	bool espconn_secure_set_size (uint8 level, uint16 size)
	uint8 level: set buffer for ESP8266 SSL server/client:
	- 0x01: SSL client;
Parameter	- 0x02: SSL server;
	- 0x03: both SSL client and SSL server
	• uint16 size: buffer size; range: 1 ~ 8192; unit: byte, default to be 2048.
Return	true: Success
	false: Failure
Note	The default buffer size is 2 KB. Please call this API before espconn_secure_accept (ESP8266 as TCP SSL server) or espconn_secure_connect (ESP8266 as TCP SSL client) to change buffer size.



4.2.25. espconn_secure_get_size

Function	Get buffer size of encrypted data (SSL).
Prototype	sint16 espconn_secure_get_size (uint8 level)
Parameter	 uint8 level: buffer for ESP8266 SSL server/client: 0x01: SSL client; 0x02: SSL server; 0x03: both SSL client and SSL server
Return	buffer size

4.2.26. espconn_secure_connect

Function	Securely connect (SSL) to a server (ESP8266 acts as a client).
Prototype	sint8 espconn_secure_connect (struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection
	0: Success
	Otherwise: Error code
Return	ESPCONN_MEM: Out of memory.
	ESPCONN_ISCONN: Already connected.
	ESPCONN_ARG: illegal argument; cannot find connection according to structure espconn.
	If espconn_secure_connect fails, a otherwise value will be returned. SSL connection fails and and therefore the ESP8266 will not enter any espconn callback.
	Only one connection is allowed when the ESP8266 acts as a SSL client. This API can be called only once. Users can call espconn_secure_disconnect to disconnect first before calling this API to create another SSL connection.
	If the SSL encrypted packet size is larger than the ESP8266 SSL buffer size (2 KB by default, set by espconn_secure_set_size), the SSL connection will fail. The ESP8266 will enter espconn_reconnect_callback
	SSL-related APIs named as espconn_secure_XXX are different from normal TCP APIs and must not be used interchangeably. In SSL connection, only espconn_secure_XXX APIs, espconn_regist_XXXcb APIs and espconn_port can be used.

Espressif 122/166 2018.08



4.2.27. espconn_secure_send

Function	Send encrypted data (SSL).
Prototype	<pre>sint8 espconn_secure_send (struct espconn *espconn, uint8 *psent, uint16 length)</pre>
Parameter	struct espconn *espconn: corresponding network connection. uint8 *psent: sent data pointer. uint16 length: sent data length.
Return	0: Success Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find SSL connection according to structure espconn.
Note	 Please call espconn_secure_send after espconn_sent_callback of the prepacket. The unencrypted data can be 1024 bytes at most per packet; the encrypted data can be 1460 bytes at most per packet.

4.2.28. espconn_secure_sent

[@deprecated] This API is deprecated; please use espconn_secure_send instead.

Function	Send encrypted data (SSL).
Prototype	<pre>sint8 espconn_secure_send (struct espconn *espconn, uint8 *psent, uint16 length)</pre>
Parameter	struct espconn *espconn: corresponding network connection uint8 *psent: sent data pointer uint16 length: sent data length
Return	0: Success Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find SSL connection according to structure espconn.
Note	 Please call espconn_secure_send after espconn_sent_callback of the prepacket. The unencrypted data can be 1024 bytes at most per packet; the encrypted data can be 1460 bytes at most per packet.

Espressif 123/166 2018.08



4.2.29. espconn_secure_disconnect

Function	Secure disconnection(SSL).
Prototype	sint8 espconn_secure_disconnect(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network connection
Return	0: Success
	Otherwise: Error code ESPCONN_ARG: illegal argument; cannot find SSL connection according to structure espconn.
Note	Do not call this API in any espconn callback. If needed, please use system_os_task and system_os_post to trigger espconn_secure_disconnect.

4.2.30. espconn_secure_ca_enable

Function	Enable SSL CA (certificate authenticate) function.
Prototype	bool espconn_secure_ca_enable (uint8 level, uint32 flash_sector)
Parameter	 uint8 level: set configuration for ESP8266 SSL server/client: 0x01: SSL client; 0x02: SSL server; 0x03: both SSL client and SSL server. uint32 flash_sector: flash sector in which CA (esp_ca_cert.bin) is downloaded. For example, if the flash_sector is 0x7B, then esp_ca_cert.bin must be downloaded to flash at 0x7B000.
Return	true: Success false: Failure
Note	 CA function is disabled by default. For more information please see <u>ESP8266</u> SDK SSL User_Manual. This API must be called before <u>espconn_secure_accept</u> (when the ESP8266 acts as TCP SSL server) or <u>espconn_secure_connect</u> (when the ESP8266 acts as TCP SSL client).

4.2.31. espconn_secure_ca_disable

Function	Disable SSL CA (certificate authenticate) function.
Prototype	bool espconn_secure_ca_disable (uint8 level)
Parameter	uint8 level: set configuration for ESP8266 SSL server/client:
	• 0x01: SSL client;
	0x02: SSL server;
	0x03: both SSL client and SSL server.

Espressif 124/166 2018.08



Return	true: Success
	false: Failure
Note	CA function is disabled by default. For more information please see ESP8266 SDK SSL User_Manual.
	This API must be called before espconn_secure_accept (when the ESP8266 acts as TCP SSL server) or espconn_secure_connect (when the ESP8266 acts as TCP SSL client).

4.2.32. espconn_secure_cert_req_enable

Function	Enable certification verification function when the ESP8266 runs as SSL client.
Prototype	bool espconn_secure_cert_req_enable (uint8 level, uint32 flash_sector)
Parameter	uint8 level: can only be set as 0x01 when ESP8266 runs as SSL client; uint32 flash_sector: set the address where secure key (esp_cert_private_key.bin) will be written in the flash. For example, parameters 0x7A should be written into address 0x7A000 in the flash. Please note that the secure key written into flash must not overlap with code binaries or system parameter binaries in the flash memory.
Return	true: Success false: Failure
Note	 Certification verification function is disabled by defaults. If the SSL server does not require certification verification, this API need not be called. This API must be called before espconn_secure_connect is called.

4.2.33. espconn_secure_cert_req_disable

Function	Disable certification verification function when ESP8266 runs as SSL client.
Prototype	bool espconn_secure_ca_disable (uint8 level)
Parameter	uint8 level: can only be set as 0x01 when ESP8266 runs as SSL client.
Return	true: Success
	false: Failure
Note	Certification verification function is disabled by defaults.

Espressif 125/166 2018.08



4.2.34. espconn_secure_set_default_certificate

Function	Set the certificate when ESP8266 runs as SSL server.
Prototype	bool espconn_secure_set_default_certificate (const uint8_t* certificate, uint16_t length)
Parameter	<pre>const uint8_t* certificate: pointer to the certificate; uint16_t length: length of the certificate.</pre>
Return	true: Success false: Failure
Note	 Demos can be found in <i>ESP8266_NONOS_SDK/examples/IoT_Demo</i>; This API has to be called before espconn_secure_accept.

4.2.35. espconn_secure_set_default_private_key

Function	Set the secure key when ESP8266 runs as SSL server.
Prototype	bool espconn_secure_set_default_private_key (const uint8_t* key, uint16_t length)
Parameter	const uint8_t* certificate: pointer to the certificate; uint16_t length: length of the certificate.
Return	true: Success false: Failure
Note	 Demos can be found in ESP8266_NONOS_SDK/examples/IoT_Demo; This API has to be called before espconn_secure_accept.

4.3. UDP APIs

4.3.1. espconn_create

Function	Create UDP transmission.
Prototype	sin8 espconn_create(struct espconn *espconn)
Parameter	struct espconn *espconn: corresponding network transmission.
Return	0: Success
	Otherwise: Error code
	ESPCONN_ISCONN: Already connected.
	ESPCONN_MEM: Out of memory.
	 ESPCONN_ARG: illegal argument, cannot find UDP transmission according to structure espconn.

Espressif 126/166 2018.08



er remote_ip and remote_port need to be set. Do not set them to be 0.	Note
---	------

4.3.2. espconn_sendto

Function	Send UDP data.
Prototype	<pre>sin16 espconn_sendto(struct espconn *espconn, uint8 *psent, uint16 length)</pre>
Parameter	struct espconn *espconn: corresponding network transmission uint8 *psent: pointer of data uint16 length: data length
Return	O: Success Otherwise: Error code • ESPCONN_ISCONN: Already connected. • ESPCONN_MEM: Out of memory. • ESPCONN_ARG: illegal argument, cannot find UDP transmission according to structure espconn.

4.3.3. espconn_igmp_join

Function	Join a multicast group.
Prototype	sint8 espconn_igmp_join(ip_addr_t *host_ip, ip_addr_t *multicast_ip)
Parameter	ip_addr_t *host_ip: IP of host
	ip_addr_t *multicast_ip: IP of multicast group
Return	0: Success
	Otherwise: Error code ESPCONN_MEM: Out of memory.
Note	This API can only be called after the ESP8266 Station connects to a router.

4.3.4. espconn_igmp_leave

Function	Quit a multicast group.
Prototype	sint8 espconn_igmp_leave(ip_addr_t *host_ip, ip_addr_t *multicast_ip)
Parameter	ip_addr_t *host_ip: IP of host
	ip_addr_t *multicast_ip: IP of multicast group
Return	0: Success
	Otherwise: Error code ESPCONN_MEM: Out of memory.

Espressif 127/166 2018.08



4.3.5. espconn_dns_setserver

Function	Set default DNS server. Two DNS servers are allowed to be set.
Prototype	void espconn_dns_setserver(uint8 numdns, ip_addr_t *dnsserver)
Parameter	uint8 numdns: DNS server ID, 0 or 1
	ip_addr_t *dnsserver: DNS server IP
Return	none
Note	Only when ESP8266 DHCP client is disabled (wifi_station_dhcpc_stop) can this API be used.

4.3.6. espconn_dns_getserver

Function	Get DNS server IP.
Prototype	ip_addr_t espconn_dns_getserver(uint8 numdns)
Parameter	uint8 numdns: DNS server ID, 0 or 1
Return	DNS server IP

Espressif 128/166 2018.08



4.4. mDNS APIs

4.4.1. espconn_mdns_init

Function	mDNS initialization
Structure	<pre>struct mdns_info{ char *host_name; char *server_name; uint16 server_port; unsigned long ipAddr; char *txt_data[10]; };</pre>
Prototype	<pre>void espconn_mdns_init(struct mdns_info *info)</pre>
Parameter	struct mdns_info *info: mDNS information
Return	none
Example	 In SoftAP + Station mode, call wifi_set_broadcast_if (STATIONAP_MODE); first to enable broadcast for both SoftAP and Station interface. Using Station interface, please obtain IP address of the ESP8266 Station first before calling the API to initialize mDNS; txt_data has to be set as "key = value".

4.4.2. espconn_mdns_close

Function	Close mDNS, corresponding creation API: espconn_mdns_init.
Prototype	void espconn_mdns_close(void)
Parameter	none
Return	none

4.4.3. espconn_mdns_server_register

Function	Register mDNS server.
Prototype	void espconn_mdns_server_register(void)
Parameter	none
Return	none

Espressif 129/166 2018.08



4.4.4. espconn_mdns_server_unregister

Function	Unregister mDNS server.
Prototype	void espconn_mdns_server_unregister(void)
Parameter	none
Return	none

4.4.5. espconn_mdns_get_servername

Function	Get mDNS server name.
Prototype	char* espconn_mdns_get_servername(void)
Parameter	none
Return	server name

4.4.6. espconn_mdns_set_servername

Function	Set mDNS server name.
Prototype	void espconn_mdns_set_servername(const char *name)
Parameter	const char *name: server name.
Return	none

4.4.7. espconn_mdns_set_hostname

Function	Set mDNS host name.
Prototype	void espconn_mdns_set_hostname(char *name)
Parameter	char *name: host name.
Return	none

4.4.8. espconn_mdns_get_hostname

Function	Get mDNS host name.
Prototype	char* espconn_mdns_get_hostname(void)
Parameter	none
Return	host name.

Espressif 130/166 2018.08



4.4.9. espconn_mdns_enable

Function	Enable mDNS.
Prototype	void espconn_mdns_enable(void)
Parameter	none
Return	none

4.4.10. espconn_mdns_disable

Function	Disable mDNS, corresponding creation API: espconn_mdns_enable.
Prototype	void espconn_mdns_disable(void)
Parameter	none
Return	none

4.4.11. Example of mDNS

Please do not use special characters (for example, "." character), or use a protocol name (for example, "http"), when defining host_name and server_name for mDNS.

```
struct mdns_info info;

void user_mdns_config()
{
    struct ip_info ipconfig;
    wifi_get_ip_info(STATION_IF, &ipconfig);
    info->host_name = "espressif";
    info->ipAddr = ipconfig.ip.addr; //ESP8266 Station IP
    info->server_name = "iot";
    info->server_port = 8080;
    info->txt_data[0] = "version = now";
    info->txt_data[1] = "user1 = data1";
    info->txt_data[2] = "user2 = data2";
    espconn_mdns_init(&info);
}
```



5. Application-Related APIs

5.1. AT APIs

AT APIs can be found in */ESP8266_NONOS_SDK/include/at_custom.h*.

For AT APIs examples, refer to *ESP8266_NONOS_SDK/examples/at*.

5.1.1. at_response_ok

Function	Output 0K to AT Port (UART0).
Prototype	void at_response_ok(void)
Parameter	none
Return	none

5.1.2. at_response_error

Function	Output 0K to AT Port (UARTO).
Prototype	void at_response_ok(void)
Parameter	none
Return	none

5.1.3. at_cmd_array_regist

Function	Register user-defined AT commands. It can be called only once to register all user-defined AT commands.
Prototype	<pre>void at_cmd_array_regist (at_function * custom_at_cmd_arrar, uint32 cmd_num)</pre>
Parameter	at_function * custom_at_cmd_arrar: Array of user-defined AT commands; uint32 cmd_num: Number counts of user-defined AT commands.
Return	none
Example	For examples please refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c.

5.1.4. at_get_next_int_dec

Function	Parse int from AT command.
----------	----------------------------

Espressif 132/166 2018.08



Prototype	bool at_get_next_int_dec (char **p_src,int* result,int* err)
Parameter	• char **p_src: *p_src is the AT command that need to be parsed;
	• int* result: int number parsed from the AT command;
	• int* err:
	- 1: no number is found;
	- 3: only "-" is found.
Return	true: parser succeeds (Notes: if no number is found, it will return True, but returns error code 1);
	false: parser is unable to parse string; some probable causes are: int number is more than 10 bytes; string contains termination characters \r; string contains only
Example	For examples please refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c.

5.1.5. at_data_str_copy

Function	Parse string from AT command.
Prototype	<pre>int32 at_data_str_copy (char * p_dest, char ** p_src,int32 max_len)</pre>
	char * p_dest: string parsed from the AT command.
Parameter	char ** p_src: *p_src is the AT command that needs to be parsed.
	int32 max_len: max string length allowed.
	length of string:
Return	 >=0: Success, and returns the length of the string;
	• <0 : Failure, and returns -1.
Example	For examples please refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c.

5.1.6. at_init

Function	Initialize AT.
Prototype	void at_init (void)
Parameter	none
Return	none
Example	For examples please refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c.

Espressif 133/166 2018.08



5.1.7. at_port_print

Function	Output string to AT port (UARTO).
Prototype	void at_port_print(const char *str)
Parameter	const char *str: string that need to output.
Return	none
Example	For examples please refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c.

5.1.8. at_set_custom_info

Function	User-defined version info of AT which can be got by AT+GMR.
Prototype	void at_set_custom_info (char *info)
Parameter	char *info: version info.
Return	none
Example	For examples please refer to ESP8266_NONOS_SDK/examples/at/user/user_main.c.

5.1.9. at_enter_special_state

Function	Enter processing state. In processing state, AT core will return busy for any further AT commands.
Prototype	void at_enter_special_state (void)
Parameter	none
Return	none

5.1.10. at_leave_special_state

Function	Exit from AT processing state.
Prototype	void at_leave_special_state (void)
Parameter	none
Return	none

Espressif 134/166 2018.08



5.1.11. at_get_version

Function	Get Espressif AT lib version.
Prototype	uint32 at_get_version (void)
Parameter	none
Return	Espressif AT lib version.

5.1.12. at_register_uart_rx_intr

Function	Set UART0 to be used by user or AT commands.
Prototype	void at_register_uart_rx_intr(at_custom_uart_rx_intr rx_func)
Parameter	at_custom_uart_rx_intr: register a UARTO Rx interrupt handler so that UARTO can be used by the customer; but if it is NULL, UARTO is assigned to AT commands.
Return	none
Example	<pre>void user_uart_rx_intr(uint8* data, int32 len) {</pre>
Note	This API can be called multiple times.When running AT, UART0 is used by AT commands by default.

5.1.13. at_response

Function	Set AT response.
Prototype	void at_response (const char *str)
Parameter	const char *str: string
Return	none



Note	at_response outputs from UART0 Tx by default, which is same as at_port_print.
	On calling at_register_response_func, the string of at_response will be the parameter of response_func.

5.1.14. at_register_response_func

Function	Register callback of at_response for user-defined responses. After calling at_register_response_func, the string of at_response will be the parameter of response_func. Users can define this behavior.
Prototype	<pre>void at_register_response_func (at_custom_response_func_type response_func)</pre>
Parameter	at_custom_response_func_type: callback of at_response.
Return	none

5.1.15. at_fake_uart_enable

Function	Enable UART simulation, which can be used to develop AT commands through SDIO or network.
Prototype	bool at_fake_uart_enable(bool enable, at_fake_uart_tx_func_type func)
Parameter	bool enable: enable UART simulation.
	at_fake_uart_tx_func_type func: callback for UART Tx simulation.
Return	true: Success
	false: Failure

5.1.16. at_fake_uart_rx

Function	UART Rx; can be used to develop AT commands through SDIO or network.
Prototype	uint32 at_fake_uart_rx(uint8* data, uint32 length)
Parameter	uint8* data: data for UART(simulation) Rx;
	uint32 length: length of data.
Return	If successful, the return value will be equal to length, otherwise, the API fails to perform its function.

5.1.17. at_set_escape_character

Function	Set an escape character for AT commands. The default escape character is \.
Prototype	bool at_set_escape_character(uint8 ch)
Parameter	uint8 ch: escape character, can be character!, or #, or \$, or @, or &, or \.

Espressif 136/166 2018.08



Return	true: Success
	false: Failure

5.2. Related JSON APIs

Found in *ESP8266_NONOS_SDK/include/json/jsonparse.h & jsontree.h*.

5.2.1. jsonparse_setup

Function	Initialize JSON parser.
Prototype	<pre>void jsonparse_setup(struct jsonparse_state *state, const char *json, int len)</pre>
Parameter	struct jsonparse_state *state: JSON parsing pointer. const char *json: JSON parsing character string. int len: character string length.
Return	none

5.2.2. jsonparse_next

Function	Returns next object of JSONparse.
Prototype	int jsonparse_next(struct jsonparse_state *state)
Parameter	struct jsonparse_state *state: JSON parsing pointer.
Return	int: parsing result.

5.2.3. jsonparse_copy_value

Function	Copies current parsing character string to a certain buffer.
Prototype	<pre>int jsonparse_copy_value(struct jsonparse_state *state, char *str, int size)</pre>
Parameter	<pre>struct jsonparse_state *state: JSON parsing pointer. char *str: buffer pointer. int size: buffer size.</pre>
Return	int: copy result.

Espressif 137/166 2018.08



5.2.4. jsonparse_get_value_as_int

Function	Parse JSON to get integer.
Prototype	<pre>int jsonparse_get_value_as_int(struct jsonparse_state *state)</pre>
Parameter	struct jsonparse_state *state: JSON parsing pointer.
Return	int: parsing result.

5.2.5. jsonparse_get_value_as_long

Function	Parse JSON to get long integer.
Prototype	long jsonparse_get_value_as_long(struct jsonparse_state *state)
Parameter	struct jsonparse_state *state: JSON parsing pointer.
Return	long: parsing result.

5.2.6. jsonparse_get_len

Function	Get parsed JSON length.
Prototype	long jsonparse_get_value_as_long(struct jsonparse_state *state)
Parameter	struct jsonparse_state *state: JSON parsing pointer.
Return	long: parsing JSON result.

5.2.7. jsonparse_get_value_as_type

Function	Parse JSON data type.
Prototype	<pre>int jsonparse_get_value_as_type(struct jsonparse_state *state)</pre>
Parameter	struct jsonparse_state *state: JSON parsing pointer.
Return	int: parsed JSON data type.

5.2.8. jsonparse_strcmp_value

Function	Compare parsed JSON and certain character string.
Prototype	<pre>int jsonparse_strcmp_value(struct jsonparse_state *state, const char *str)</pre>
Parameter	struct jsonparse_state *state: JSON parsing pointer;
	const char *str: character buffer.
Return	int: comparison result.

Espressif 138/166 2018.08



5.2.9. jsontree_set_up

Function	Create JSON data tree.
Prototype	<pre>void jsontree_setup(struct jsontree_context *js_ctx, struct jsontree_value *root, int (* putchar)(int))</pre>
Parameter	<pre>struct jsontree_context *js_ctx: JSON data tree pointer; struct jsontree_value *root: root element pointer; int (* putchar)(int): input function.</pre>
Return	none

5.2.10. jsontree_reset

Function	Resets JSON tree.
Prototype	void jsontree_reset(struct jsontree_context *js_ctx)
Parameter	struct jsontree_context *js_ctx: JSON data tree pointer;
Return	none

5.2.11. jsontree_path_name

Function	Get JSON tree parameters.
Prototype	<pre>const char *jsontree_path_name(</pre>
Parameter	<pre>struct jsontree_context *js_ctx: JSON data tree pointer; int depth: JSON tree depth</pre>
Return	char*: parameter pointer

5.2.12. jsontree_write_int

Function	Write integer to JSON tree.
Prototype	<pre>void jsontree_write_int(</pre>

Espressif 139/166 2018.08



Parameter	<pre>struct jsontree_context *js_ctx: JSON data tree pointer; int value: integer value</pre>
Return	none

5.2.13. jsontree_write_int_array

Function	Write integer array to JSON tree.
Prototype	<pre>void jsontree_write_int_array(</pre>
Parameter	<pre>struct jsontree_context *js_ctx: JSON data tree pointer; int *text: array entry address; uint32 length: array length.</pre>
Return	none

5.2.14. jsontree_write_string

Function	Writes string to JSON tree.
Prototype	<pre>void jsontree_write_string(</pre>
Parameter	<pre>struct jsontree_context *js_ctx: JSON data tree pointer; const char* text: character string pointer.</pre>
Return	none

5.2.15. jsontree_print_next

Function	JSON tree depth.
Prototype	<pre>int jsontree_print_next(struct jsontree_context *js_ctx)</pre>
Parameter	struct jsontree_context *js_ctx: JSON data tree pointer.
Return	JSON tree depth.

Espressif 140/166 2018.08



5.2.16. jsontree_find_next

Function	Find JSON tree element.
Prototype	<pre>struct jsontree_value *jsontree_find_next(struct jsontree_context *js_ctx, int type)</pre>
Parameter	<pre>struct jsontree_context *js_ctx: JSON data tree pointer; int: type.</pre>
Return	struct jsontree_value *: JSON tree element pointer.



6. Definitions & Structures

6.1. Timer

6.2. Wi-Fi-Related Structures

6.2.1. Station Parameters

Notice:

BSSID is the MAC address of AP, which will be used when several APs have the same SSID. If station_config.bssid_set==1, station_config.bssid has to be set; otherwise, the connection will fail. In general, station_config.bssid_set need to be 0.

6.2.2. SoftAP Parameters

```
typedef enum _auth_mode {
   AUTH_OPEN = 0,
   AUTH_WEP,
```



```
AUTH_WPA_PSK,
   AUTH_WPA2_PSK,
   AUTH_WPA_WPA2_PSK
} AUTH_MODE;
struct softap_config {
   uint8 ssid[32];
   uint8 password[64];
   uint8 ssid_len;
                           // support 1 ~ 13
   uint8 channel;
                           // Don't support AUTH_WEP in SoftAP mode
   uint8 authmode;
   uint8 ssid_hidden;
                           // default 0
   uint8 max_connection; // default 4, max 4
   uint16 beacon_interval; // 100 ~ 60000 ms, default 100
};
```

! Notice:

If softap_config.ssid_len==0, SSID is checked till a termination character is found; otherwise, set the length of SSID according to softap_config.ssid_len.

6.2.3. Scan Parameters

```
struct scan_config {
    uint8 *ssid;
    uint8 *bssid;
    uint8 channel;
    uint8 show_hidden; // Scan APs which are hiding their SSID or not.
    wifi_scan_type_t scan_type; // scan type, active or passive
    wifi_scan_time_t scan_time; // scan time per channel
};
struct bss_info {
    STAILQ_ENTRY(bss_info)
                             next;
    uint8 bssid[6];
    uint8 ssid[32];
    uint8 ssid_len;
    uint8 channel;
    sint8 rssi;
    AUTH_MODE authmode;
    uint8 is_hidden; // SSID of current AP is hidden or not.
                              // AP' s frequency offset
    sint16 freq_offset;
    sint16 freqcal_val;
    uint8 *esp_mesh_ie;
    uint8 simple_pair;
    CIPHER_TYPE pairwise_cipher;
    CIPHER_TYPE group_cipher;
    uint32_t phy_11b:1;
    uint32_t phy_11g:1;
    uint32_t phy_11n:1;
    uint32_t wps:1;
```



```
uint32_t reserved:28;
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```

6.2.4. Wi-Fi Event-Related Structures

```
enum {
    EVENT_STAMODE_CONNECTED = 0,
    EVENT_STAMODE_DISCONNECTED,
    EVENT_STAMODE_AUTHMODE_CHANGE,
    EVENT_STAMODE_GOT_IP,
   EVENT_STAMODE_DHCP_TIMEOUT,
    EVENT_SOFTAPMODE_STACONNECTED,
    EVENT_SOFTAPMODE_STADISCONNECTED,
    EVENT_SOFTAPMODE_PROBEREQRECVED,
    EVENT_OPMODE_CHANGED,
    EVENT_SOFTAPMODE_DISTRIBUTE_STA_IP
    EVENT_MAX
};
enum {
       REASON_UNSPECIFIED
                                      = 1,
       REASON_AUTH_EXPIRE
                                      = 2,
       REASON_AUTH_LEAVE
                                      = 3.
       REASON_ASSOC_EXPIRE
       REASON_ASSOC_TOOMANY
                                      = 5.
       REASON_NOT_AUTHED
                                      = 6,
       REASON_NOT_ASSOCED
                                      = 7,
       REASON_ASSOC_LEAVE
                                     = 9,
       REASON_ASSOC_NOT_AUTHED
                                      = 10, /* 11h */
       REASON_DISASSOC_PWRCAP_BAD
       REASON_DISASSOC_SUPCHAN_BAD = 11, /* 11h */
                                      = 13, /* 11i */
       REASON_IE_INVALID
       REASON_MIC_FAILURE
                                      = 14, /* 11i */
       REASON_4WAY_HANDSHAKE_TIMEOUT = 15, /* 11i */
       REASON_GROUP_KEY_UPDATE_TIMEOUT = 16, /* 11i */
                                     = 17. /* 11i */
       REASON_IE_IN_4WAY_DIFFERS
       REASON_GROUP_CIPHER_INVALID
                                      = 18, /* 11i */
       REASON_PAIRWISE_CIPHER_INVALID = 19, /* 11i */
                                      = 20, /* 11i */
       REASON_AKMP_INVALID
       REASON_UNSUPP_RSN_IE_VERSION = 21, /* 11i */
                                    = 22, /* 11i */
       REASON_INVALID_RSN_IE_CAP
                                     = 23, /* 11i */
       REASON_802_1X_AUTH_FAILED
       REASON_CIPHER_SUITE_REJECTED = 24, /* 11i */
       REASON_BEACON_TIMEOUT
                                      = 200,
                                      = 201,
       REASON_NO_AP_FOUND
       REASON_AUTH_FAIL
                                      = 202,
       REASON_ASSOC_FAIL
                                      = 203,
       REASON_HANDSHAKE_TIMEOUT
                                       = 204,
```



```
};
typedef struct {
        uint8 ssid[32];
        uint8 ssid_len;
        uint8 bssid[6];
        uint8 channel;
} Event_StaMode_Connected_t;
typedef struct {
        uint8 ssid[32];
        uint8 ssid_len;
        uint8 bssid[6];
        uint8 reason;
} Event_StaMode_Disconnected_t;
typedef struct {
        uint8 old_mode;
        uint8 new_mode;
} Event_StaMode_AuthMode_Change_t;
typedef struct {
        struct ip_addr ip;
        struct ip_addr mask;
        struct ip_addr gw;
} Event_StaMode_Got_IP_t;
typedef struct {
        uint8 mac[6];
        uint8 aid;
} Event_SoftAPMode_StaConnected_t;
typedef struct {
        uint8 mac[6];
        struct ip_addr ip;
        uint8 aid;
} Event_SoftAPMode_Distribute_Sta_IP_t;
typedef struct {
        uint8 mac[6];
        uint8 aid;
} Event_SoftAPMode_StaDisconnected_t;
typedef struct {
        int rssi;
        uint8 mac[6];
} Event_SoftAPMode_ProbeReqRecved_t;
typedef struct {
        uint8 old_opmode;
```



```
uint8 new_opmode;
} Event_OpMode_Change_t;
typedef union {
        Event_StaMode_Connected_t
                                                 connected;
        Event_StaMode_Disconnected_t
                                                 disconnected;
        Event_StaMode_AuthMode_Change_t
                                                 auth_change;
        Event_StaMode_Got_IP_t
                                                 got_ip;
        Event_SoftAPMode_StaConnected_t
                                                 sta_connected;
        Event_SoftAPMode_Distribute_Sta_IP_t
                                                 distribute_sta_ip;
        Event_SoftAPMode_StaDisconnected_t
                                                 sta_disconnected;
        Event_SoftAPMode_ProbeReqRecved_t
                                                 ap_probereqrecved;
        Event_OpMode_Change_t
                                                 opmode_changed;
} Event_Info_u;
typedef struct _esp_event {
    uint32 event;
    Event_Info_u event_info;
} System_Event_t;
```

6.2.5. SmartConfig Structures

6.3. JSON-Related Structure

6.3.1. JSON Structures

```
struct jsontree_value {
    uint8_t type;
};

struct jsontree_pair {
    const char *name;
    struct jsontree_value *value;
};

struct jsontree_context {
    struct jsontree_value *values[JSONTREE_MAX_DEPTH];
```



```
uint16_t index[JSONTREE_MAX_DEPTH];
    int (* putchar)(int);
    uint8_t depth;
    uint8_t path;
    int callback_state;
};
struct jsontree_callback {
    uint8_t type;
    int (* output)(struct jsontree_context *js_ctx);
    int (* set)(struct jsontree_context *js_ctx,
              struct jsonparse_state *parser);
};
struct jsontree_object {
    uint8_t type;
    uint8_t count;
    struct jsontree_pair *pairs;
};
struct jsontree_array {
    uint8_t type;
    uint8_t count;
    struct jsontree_value **values;
};
struct jsonparse_state {
    const char *json;
    int pos;
    int len;
    int depth;
    int vstart;
    int vlen;
    char vtype;
    char error;
    char stack[JSONPARSE_MAX_DEPTH];
};
             JSON Macro Definitions
#define JSONTREE_OBJECT(name, ...)
static struct jsontree_pair jsontree_pair_##name[] = {__VA_ARGS__};
static struct jsontree_object name = {
    JSON_TYPE_OBJECT,
sizeof(jsontree_pair_##name)/sizeof(struct jsontree_pair),
    jsontree_pair_##name }
#define JSONTREE_PAIR_ARRAY(value) (struct jsontree_value *)(value)
#define JSONTREE_ARRAY(name, ...)
static struct jsontree_value* jsontree_value_##name[] = {__VA_ARGS__};
static struct jsontree_array name = {
    JSON_TYPE_ARRAY,
    sizeof(jsontree_value_##name)/sizeof(struct jsontree_value*),
```



jsontree_value_##name }

6.4. espconn Parameters

6.4.1. Callback Functions

```
/** callback prototype to inform about events for a espconn */
typedef void (* espconn_recv_callback)(void *arg, char *pdata, unsigned short len);
typedef void (* espconn_callback)(void *arg, char *pdata, unsigned short len);
typedef void (* espconn_connect_callback)(void *arg);
```

6.4.2. espconn Structures

```
typedef void* espconn_handle;
typedef struct _esp_tcp {
    int remote_port;
    int local_port;
    uint8 local_ip[4];
    uint8 remote_ip[4];
        espconn_connect_callback connect_callback;
        espconn_reconnect_callback reconnect_callback;
        espconn_connect_callback disconnect_callback;
        espconn_connect_callback write_finish_fn;
} esp_tcp;
typedef struct _esp_udp {
    int remote_port;
    int local_port;
    uint8 local_ip[4];
    uint8 remote_ip[4];
} esp_udp;
/** Protocol family and type of the espconn */
enum espconn_type {
    ESPCONN_INVALID
    /* ESPCONN_TCP Group */
    ESPCONN_TCP
                       = 0 \times 10,
    /* ESPCONN_UDP Group */
    ESPCONN_UDP
                       = 0x20,
};
/** Current state of the espconn. Non-TCP espconn are always in state ESPCONN_NONE! */
enum espconn_state {
    ESPCONN_NONE,
    ESPCONN_WAIT,
    ESPCONN_LISTEN,
    ESPCONN_CONNECT,
    ESPCONN_WRITE,
```



```
ESPCONN_READ,
    ESPCONN_CLOSE
};
enum espconn_option{
        ESPCONN\_START = 0x00,
        ESPCONN_REUSEADDR = 0x01,
        ESPCONN_NODELAY = 0x02,
        ESPCONN\_COPY = 0x04,
        ESPCONN_KEEPALIVE = 0x08,
        ESPCONN\_MANUALRECV = 0x10,
        ESPCONN_END
}
enum espconn_level{
        ESPCONN_KEEPIDLE,
        ESPCONN_KEEPINTVL,
        ESPCONN_KEEPCNT
/** A espconn descriptor */
struct espconn {
    /** type of the espconn (TCP, UDP) */
    enum espconn_type type;
    /** current state of the espconn */
    enum espconn_state state;
    union {
        esp_tcp *tcp;
        esp_udp *udp;
    } proto;
    /** A callback function that is informed about events for this espconn */
    espconn_recv_callback recv_callback;
    espconn_sent_callback sent_callback;
    uint8 link_cnt;
    void *reverse; // reversed for customer use
};
```

6.4.3. Interrupt-Related Definitions

```
/* interrupt related */
#define ETS_SPI_INUM 2
#define ETS_GPIO_INUM 4
#define ETS_UART_INUM 5
#define ETS_UART1_INUM 5
#define ETS_FRC_TIMER1_INUM 9

/* disable all interrupts */
#define ETS_INTR_LOCK() ets_intr_lock()
/* enable all interrupts */
#define ETS_INTR_UNLOCK() ets_intr_unlock()
```



```
/* register interrupt handler of frc timer1 */
#define ETS_FRC_TIMER1_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_FRC_TIMER1_INUM, (func), (void *)(arg))
/* register interrupt handler of GPIO */
#define ETS_GPIO_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_GPIO_INUM, (func), (void *)(arg))
/* register interrupt handler of UART */
#define ETS_UART_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_UART_INUM, (func), (void *)(arg))
/* register interrupt handler of SPI */
#define ETS_SPI_INTR_ATTACH(func, arg) \
ets_isr_attach(ETS_SPI_INUM, (func), (void *)(arg))
/* enable a interrupt */
#define ETS_INTR_ENABLE(inum) ets_isr_unmask((1<<inum))</pre>
/* disable a interrupt */
#define ETS_INTR_DISABLE(inum) ets_isr_mask((1<<inum))</pre>
/* enable SPI interrupt */
#define ETS_SPI_INTR_ENABLE() ETS_INTR_ENABLE(ETS_SPI_INUM)
/* enable UART interrupt */
#define ETS_UART_INTR_ENABLE() ETS_INTR_ENABLE(ETS_UART_INUM)
/* disable UART interrupt */
#define ETS_UART_INTR_DISABLE() ETS_INTR_DISABLE(ETS_UART_INUM)
/* enable frc1 timer interrupt */
#define ETS_FRC1_INTR_ENABLE() ETS_INTR_ENABLE(ETS_FRC_TIMER1_INUM)
/* disable frc1 timer interrupt */
#define ETS_FRC1_INTR_DISABLE() ETS_INTR_DISABLE(ETS_FRC_TIMER1_INUM)
/* enable GPIO interrupt */
#define ETS_GPIO_INTR_ENABLE() ETS_INTR_ENABLE(ETS_GPIO_INUM)
/* disable GPIO interrupt */
#define ETS_GPIO_INTR_DISABLE() ETS_INTR_DISABLE(ETS_GPIO_INUM)
```



7. Peripheral-Related Drivers

For peripheral drivers please see /ESP8266_NONOS_SDK/driver_lib.

7.1. GPIO Related APIs

GPIO APIs can be found in /ESP8266_NONOS_SDK/include/eagle_soc.h & gpio.h.

Please refer to /ESP8266_NONOS_SDK/examples/IoT_Demo/user/user_plug.c.

7.1.1. PIN-Related Macros

The following macros are used to control the GPIO pins' status.

PIN_PULLUP_DIS(PIN_NAME)		Disable pin pull-up.	Example:
PIN_PULLUP_EN(PIN_NAME)		Enable pin pull up.	// Use MTDI pin as GPI012.
PIN_F	UNC_SELECT(PIN_NAME, FUNC)	Select pin function.	FUNC_GPI012);

7.1.2. gpio_output_set

Function	Set GPIO property.	
Prototype	<pre>void gpio_output_set(uint32 set_mask, uint32 clear_mask, uint32 enable_mask, uint32 disable_mask)</pre>	
Parameter	uint32 set_mask: set high output; 1: high output; 0: no status change; uint32 clear_mask: set low output; 1: low output; 0: no status change; uint32 enable_mask: enable output bit; uint32 disable_mask: enable input bit.	
Return	none	
Example	<pre>gpio_output_set(BIT12, 0, BIT12, 0): Set GPIO12 as high-level output. gpio_output_set(0, BIT12, BIT12, 0): Set GPIO12 as low-level output. gpio_output_set(BIT12, BIT13, BIT12 BIT13, 0): Set GPIO12 as high-level output, and GPIO13 as low-level output. gpio_output_set(0, 0, 0, BIT12): Set GPIO12 as input.</pre>	

Espressif 151/166 2018.08



7.1.3. GPIO input and output macros

GPIO_OUTPUT_SET(gpio_no, bit_value)	Set gpio_no as output bit_value, the same as the output example in 8.1.2.
GPIO_DIS_OUTPUT(gpio_no)	Set gpio_no as input, the same as the input example in 8.1.2.
GPIO_INPUT_GET(gpio_no)	Get the level status of gpio_no.

7.1.4. GPIO interrupt

ETS_GPIO_INTR_ATTACH(func, arg)	Register GPIO interrupt control function.
ETS_GPIO_INTR_DISABLE()	Disable GPIO interrupt.
ETS_GPIO_INTR_ENABLE()	Enable GPIO interrupt.

7.1.5. gpio_pin_intr_state_set

Function	Set GPIO interrupt state.
Prototype	<pre>void gpio_pin_intr_state_set(uint32 i, GPIO_INT_TYPE intr_state)</pre>
Parameter	<pre>uint32 i : GPIO pin ID, if you want to set GPIO14, pls use GPIO_ID_PIN(14); GPIO_INT_TYPE intr_state : interrupt type as the following: typedef enum { GPIO_PIN_INTR_DISABLE = 0, GPIO_PIN_INTR_POSEDGE = 1, GPIO_PIN_INTR_NEGEDGE = 2, GPIO_PIN_INTR_ANYEDGE = 3, GPIO_PIN_INTR_LOLEVEL = 4, GPIO_PIN_INTR_HILEVEL = 5 } GPIO_INT_TYPE;</pre>
Return	none

7.1.6. GPIO Interrupt Handler

Follow the steps below to clear interrupt status in GPIO interrupt processing function:

```
uint32 gpio_status;
gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
//clear interrupt status
GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
```



7.2. UART-Related APIs

By default, UART0 is a debug output interface. In the case of a dual UART, UART0 works as data receive and transmit interface, while UART1 acts as the debug output interface. Please make sure all hardware is correctly connected.

For detailed information on UART, please see <u>ESP8266 Technical Reference</u>.

7.2.1. uart_init

Function	Initialize baud rates of the two UARTs.
Prototype	<pre>void uart_init(UartBautRate uart0_br, UartBautRate uart1_br)</pre>
Parameter	UartBautRate uart0_br: UARTO baud rate; UartBautRate uart1_br: UART1 baud rate.
Baud rates	typedef enum { BIT_RATE_9600 = 9600, BIT_RATE_19200 = 19200, BIT_RATE_38400 = 38400, BIT_RATE_57600 = 57600, BIT_RATE_74880 = 74880, BIT_RATE_115200 = 115200, BIT_RATE_230400 = 230400, BIT_RATE_460800 = 460800, BIT_RATE_921600 = 921600 } UartBautRate;
Return	none

7.2.2. uart0_tx_buffer

Function	Send user-defined data through UART0.
Prototype	void uart0_tx_buffer(uint8 *buf, uint16 len)
Parameter	uint8 *buf: data to be sent;
	uint16 len: the length of data to be sent.
Return	none

7.2.3. uart0_rx_intr_handler

Function	UART0 interrupt processing function. Users can process the received data in this function.
Prototype	void uart0_rx_intr_handler(void *para)
Parameter	void *para: the pointer pointing to RcvMsgBuff structure.

Espressif 153/166 2018.08



rn none

7.2.4. uart_div_modify

Function	Set baud rate of UART.
Prototype	void uart_div_modify(uint8 uart_no, uint32 DivLatchValue)
Parameter	uint8 uart_no: UART number, UART0 or UART1. uint32 DivLatchValue: Clock division parameter.
Return	none
Example	<pre>void ICACHE_FLASH_ATTR UART_SetBaudrate(uint8 uart_no, uint32 baud_rate) { uart_div_modify(uart_no, UART_CLK_FREQ /baud_rate); }</pre>

7.3. I2C Master-Related APIs

7.3.1. i2c_master_gpio_init

Function	Set GPIO in I2C master mode.
Prototype	void i2c_master_gpio_init (void)
Parameter	none
Return	none

7.3.2. i2c_master_init

Function	Initialize I2C.
Prototype	void i2c_master_init(void)
Parameter	none
Return	none

7.3.3. i2c_master_start

Function	Configure I2C to start sending data.
Prototype	void i2c_master_start(void)
Parameter	none
Return	none

Espressif 154/166 2018.08



7.3.4. i2c_master_stop

Function	Configure I2C to stop sending data.
Prototype	void i2c_master_stop(void)
Parameter	none
Return	none

7.3.5. i2c_master_send_ack

Function	Sends I2C ACK.
Prototype	void i2c_master_send_ack (void)
Parameter	none
Return	none

7.3.6. i2c_master_send_nack

Function	Sends I2C NACK.
Prototype	void i2c_master_send_nack (void)
Parameter	none
Return	none

7.3.7. i2c_master_checkAck

Function	Checks the ACK from the slave.
Prototype	bool i2c_master_checkAck (void)
Parameter	none
Return	true: ACK received from I2C slave
	false: NACK received from I2C slave

7.3.8. i2c_master_readByte

Function	Read one byte from I2C slave.
Prototype	uint8 i2c_master_readByte (void)
Parameter	none
Return	uint8: the value that was read

Espressif 155/166 2018.08



7.3.9. i2c_master_writeByte

Function	Write one byte to the slave.
Prototype	void i2c_master_writeByte (uint8 wrdata)
Parameter	uint8 wrdata: data to write
Return	none

7.4. PWM-Related APIs

The document will introduce PWM-related APIs from *pwm.h*. For more information on PWM-related APIs please see *ESP8266 Technical Reference*.

PWM APIs can not be called when APIs in *hw_timer.c* are in use, because they use the same hardware timer.

Do not set the system to be Light-sleep mode (wifi_set_sleep_type(LIGT_SLEEP);), because CPU is halted and will not be interrupted by NMI during Light-sleep. To enter Deep-sleep mode, PWM needs to be stopped first.

7.4.1. pwm_init

Function	Initialize PWM function, including GPIO selection, period and duty cycle.
Prototype	<pre>void pwm_init(uint32 period, uint8 *duty, uint32 pwm_channel_num, uint32 (*pin_info_list)[3])</pre>
Parameter	uint32 period: PWM period uint8 *duty: duty cycle of each output uint32 pwm_channel_num: PWM channel number uint32 (*pin_info_list)[3]: GPIO parameter of PWM channel.It is a pointer of n * 3 array which defines GPIO register, IO reuse of corresponding PIN and GPIO number.
Return	none
Example	<pre>uint32 io_info[][3] =</pre>
Note	This API can be called only once.

Espressif 156/166 2018.08



7.4.2. pwm_start

Function	Starts PWM. This function needs to be called after PWM configuration is changed.
Prototype	void pwm_start (void)
Parameter	none
Return	none

7.4.3. pwm_set_duty

Function	Sets duty cycle of a PWM output. Set the time that high-level signal will last. The range of duty depends on PWM period. Its maximum value of which can be Period * 1000 /45. For example, for 1-KHz PWM, the duty range is 0 ~ 22222.	
Prototype	void pwm_set_duty(uint32 duty, uint8 channel)	
Parameter	uint32 duty: the time that high-level single will last, duty cycle will be (duty*45)/ (period*1000);	
	uint8 channel: PWM channel, which depends on how many PWM channels is used. In IOT_Demo it depends on #define PWM_CHANNEL.	
Return	none	

7.4.4. pwm_get_duty

Function	Get duty cycle of PWM output; duty cycle will be (duty*45)/ (period*1000).
Prototype	uint8 pwm_get_duty(uint8 channel)
Parameter	uint8 channel: PWM channel, which depends on how many PWM channels is used. In IOT_Demo it depends on #define PWM_CHANNEL.
Return	uint8: duty cycle of PWM output.

7.4.5. pwm_set_period

Function	Set PWM period, unit: μs. For example, for 1-KHz PWM, the period is 1000 μs.
Prototype	void pwm_set_period(uint32 period)
Parameter	uint32 period: PWM period, unit: µs.
Return	none
Note	After updating the configuration, pwm_start must be called for the changes to take effect.

Espressif 157/166 2018.08



7.4.6. pwm_get_period

Function	Get PWM period.
Prototype	uint32 pwm_get_period(void)
Parameter	none
Return	PWM period; unit: µs.

7.4.7. get_pwm_version

Function	Get version information of PWM.
Prototype	uint32 get_pwm_version(void)
Parameter	none
Return	PWM version

7.5. SDIO APIs

ESP8266 can only work as SDIO slave.

7.5.1. sdio_slave_init

Function	SDIO slave initialization.
Prototype	void sdio_slave_init(void)
Parameter	none
Return	none

7.5.2. sdio_load_data

Function	Load data into SDIO buffer, and inform SDIO host to read it.
Prototype	int32 sdio_load_data(const uint8* data, uint32 len)
Parameter	const uint8* data: data that will be transmitted; uint32 len: the length of data.
Return	The length of data that be loaded successfully. If the data length is too long to fit in SDIO buffer, this API will return 0, which means it failed to load data.

Espressif 158/166 2018.08



7.5.3. sdio_register_recv_cb

Function	Register a callback which will be called when ESP8266 receives data from the host through SDIO.	
Callback function	typedef void(*sdio_recv_data_callback)(uint8* data, uint32 len)	
	The sdio_recv_data_callback can not be stored in cache, so please do not define ICACHE_FLASH_ATTR before it.	
Callback Definition	bool sdio_register_recv_cb(sdio_recv_data_callback cb)	
Parameter	sdio_recv_data_callback cb: callback	
Return	true: Success	
	false: Failure	



Α.

Appendix

A.1. ESPCONN Programming

For Espressif sample codes, please see:

https://github.com/espressif/esp8266-nonos-sample-code

https://github.com/espressif/esp8266-rtos-sample-code

A.1.1. TCP Client Mode

! Notice:

- ESP8266, working in Station mode, will start client connections when given an IP address of the AP (the router) it connects to.
- ESP8266, working in SoftAP mode, will start client connections when the devices connected to the ESP8266 are given IP addresses.

Steps:

- 1. Initialize espconn parameters according to protocols.
- 2. Register connect callback function and reconnect callback function.
 - Call espconn_regist_connectcb and espconn_regist_reconcb.
- 3. Call espconn_connect function and set up connection with TCP Server.
- Registered connected callback functions will be called after successful connection, which will register corresponding callback function. We recommend registering a disconnect callback function.
 - Call espconn_regist_recvcb, espconn_regist_sentcb and espconn_regist_disconcb in connected callback.
- 5. When using callback function of received data or callback function of sent data to close connection, it is recommended to set a time delay to make sure that the all firmware functions are completed.

A.1.2. TCP Server Mode

! Notice:

- If the ESP8266 is in Station mode, it will start server listening when given an IP address.
- If the ESP8266 is in SoftAP mode, it will start server listening directly.

Steps:

- 1. Initialize espconn parameters according to protocols.
- 2. Register connect callback function and reconnect callback function.



- Call espconn_regist_connectcb and espconn_regist_reconcb.
- 3. Call espconn_accept to listen to the connection with host...
- 4. Registered connect function will be called after a successful connection, which will register a corresponding callback function.
 - Call espconn_regist_recvcb, espconn_regist_sentcb and espconn_regist_disconcb in connected callback.

A.1.3. espconn callback

Register Function	Callback	Description
espconn_regist_connectcb	espconn_connect_callback	TCP connected successfully.
espconn_regist_reconcb	espconn_reconnect_callback	TCP connected successfully.
espconn_regist_sentcb	espconn_sent_callback	TCP or UDP data is successfully sent.
espconn_regist_recvcb	espconn_recv_callback	Received TCP or UDP data.
espconn_regist_write_finish	espconn_write_finish_callback	Write data into TCP-send-buffer.
espconn_regist_disconcb	espconn_disconnect_callback	TCP disconnected successfully.

! Notice:

- Parameter arg of callback is the pointer corresponding structure espconn. This pointer may be different in different callbacks, please do not use this pointer directly to distinguish one from another in multiple connections. Use remote_ip and remote_port in espconn instead.
- If espconn_connect (or espconn_secure_connect) fails and returns a otherwise value, that means there is no connection, so it won't enter any espconn callback.
- Don't call espconn_disconnect (or espconn_secure_disconnect) to break the TCP connection in any espconn callback.
- If it is needed, please use system_os_task and system_os_post to trigger the disconnection (espconn_disconnect or espconn_secure_disconnect).

A.2. RTC APIs Example

Demo code below shows how to get RTC time and to read and write to RTC memory.

```
#include "ets_sys.h"
#include "osapi.h"
#include "user_interface.h"

os_timer_t rtc_test_t;
#define RTC_MAGIC 0x55aaaa55
typedef struct {
    uint64 time_acc;
    uint32 magic ;
```



```
uint32 time_base;
}RTC_TIMER_DEMO;
void rtc_count()
   RTC_TIMER_DEMO rtc_time;
    static uint8 cnt = 0;
    system_rtc_mem_read(64, &rtc_time, sizeof(rtc_time));
    if(rtc_time.magic!=RTC_MAGIC){
       os_printf("rtc time init...\r\n");
       rtc_time.magic = RTC_MAGIC;
       rtc_time.time_acc= 0;
       rtc_time.time_base = system_get_rtc_time();
       os_printf("time base : %d \r\n",rtc_time.time_base);
   }
   os_printf("======\r\n");
   os_printf("RTC time test : \r\n");
   uint32 rtc_t1,rtc_t2;
   uint32 st1,st2;
   uint32 cal1, cal2;
   rtc_t1 = system_get_rtc_time();
   st1 = system_get_time();
    cal1 = system_rtc_clock_cali_proc();
   os_delay_us(300);
   st2 = system_get_time();
   rtc_t2 = system_get_rtc_time();
    cal2 = system_rtc_clock_cali_proc();
   os_printf(" rtc_t2-t1 : %d \r\n",rtc_t2-rtc_t1);
   os_printf(" st2-t2 : %d \r\n",st2-st1);
   os_printf("cal 1 : %d.%d \r\n", ((cal1*1000)>>12)/1000, ((cal1*1000)>>12)%1000);
   os_printf("cal 2 : %d.%d \r\n",((cal2*1000)>>12)/1000,((cal2*1000)>>12)%1000 );
   os_printf("======\r\n\r\n");
    rtc_time.time_acc += ( ((uint64)(rtc_t2 - rtc_time.time_base)) * ( (uint64)
((cal2*1000)>>12)) );
   os_printf("rtc time acc : %lld \r\n",rtc_time.time_acc);
   os_printf("power on time : %lld us\r\n", rtc_time.time_acc/1000);
    os_printf("power on time : %11d.\%0211d S\r\n", (rtc_time.time_acc/10000000)/100,
(rtc_time.time_acc/10000000)%100);
    rtc_time.time_base = rtc_t2;
    system_rtc_mem_write(64, &rtc_time, sizeof(rtc_time));
   os_printf("----\r\n");
    if(5== (cnt++)){
       os_printf("system restart\r\n");
       system_restart();
```



A.3. Sniffer Introduction

For more details on sniffer, please refer to ESP8266 Technical Reference.

A.4. ESP8266 SoftAP and Station Channel Configuration

Even though ESP8266 supports the SoftAP + Station mode, it is limited to only one hardware channel.

In the SoftAP + Station mode, the ESP8266 SoftAP will adjust its channel configuration to be the same as the ESP8266 Station.

This limitation may cause some inconveniences in the SoftAP + Station mode that users need to pay special attention to, for example:

Case 1:

- (1) When users connect the ESP8266 Station to a router (for example, channel 6),
- (2) and sets the ESP8266 SoftAP through wifi_softap_set_config,
- (3) if the value set is effective, the API will return **true**. However, the channel will be automatically adjusted to channel 6 in order to be in line with the ESP8266 Station interface. This is because there is only one hardware channel in this mode.

Case 2:

- (1) If users set the channel of the ESP8266 SoftAP through wifi_softap_set_config (for example, channel 5),
- (2) other Stations will connect to the ESP8266 SoftAP.
- (3) When the users connects the ESP8266 Station to a router (for example, channel 6),
- (4) the ESP8266 SoftAP will adjust its channel to be the same as that of the ESP8266 Station (which is channel 6 in this case).
- (5) As a result of the change of channel, the Station Wi-Fi connected to the ESP8266 SoftAP in Step Two will be disconnected.

Case 3:



- (1) Other stations are connected to the ESP8266 SoftAP.
- (2) If the ESP8266's Station interface has been scanning or trying to connect to a target router, the ESP8266 SoftAP's connection may terminate.
- (3) This is because the ESP8266 Station will try to find its target router in different channels, which means it will keep changing channels, and as a result, the ESP8266 channel is changing, too. Therefore, the ESP8266 SoftAP's connection may terminate.
- (4) In cases like this, users can set a timer to call wifi_station_disconnect to stop the ESP8266 Station from continuously trying to connect to a router. Or use wifi_station_set_reconnect_policy or wifi_station_set_auto_connect to disable the ESP8266 Station from reconnecting to the router.

A.5. ESP8266 Boot Messages

ESP8266 outputs boot messages through UART0 with a baud rate of 74880.

```
ets Jan 8 2013,rst cause:2, boot mode:(3,6)

load 0x4010f000, len 1264, room 16

tail 0

chksum 0x42

csum 0x42
```

Messages	Description
rst cause	1: power on
	2: external reset
	4: hardware watchdog reset
boot mode (the first parameter)	1: ESP8266 is in UART-down mode (and downloads firmware into flash).
	3: ESP8266 is in Flash-boot mode (and boots up from flash).
chksum	If chksum == csum, it means that flash is correctly read during booting.

A.6. ESP8266 Signaling Measurement

Signaling Measurement has been supported since ESP8266_NonOS_SDK_V3.0. It is disabled by default. But you can call wifi_enable_signaling_measurement() and wifi_disable_signaling_measurement(), which are defined in user_interface.h, to enable and disable it. It is suggested to use the CMW500 tester from ROHDE&SCHWARZ for signal measurement. See the detailed steps below:



- (1) After start up, call wifi_set_opmode to enable station mode, and then call wifi_enable_signaling_measurement to enable signal measurement.
- (2) If you need to test 11n or 11b, please call wifi_set_phy_mode to set phy mode first. Otherwise, it is not necessary to call API as ESP8266 will be in 11g mode by default.
- (3) Call wifi_station_connect to connect ESP8266 to the tester.



Espressif IOT Team www.espressif.com

Disclaimer and Copyright Notice

Information in this document, including URL references, is subject to change without notice.

THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

All liability, including liability for infringement of any proprietary rights, relating to the use of information in this document, is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

The Wi-Fi Alliance Member logo is a trademark of the Wi-Fi Alliance. The Bluetooth logo is a registered trademark of Bluetooth SIG.

All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners, and are hereby acknowledged.

Copyright © 2018 Espressif Inc. All rights reserved.