Freescale MQX™ RTOS RTCS 用户指南（IPv4 和 IPv6）

Document Number: MQXRTCSUG
Rev 2, 04/2015

内容

小节编号 标题 页

第 1 章
前言

1.1 关于本手册....23
1.2 如何获取更多信息....23
1.3 约定....23
1.3.1 产品名称....23
1.3.2 提示....23
1.3.3 附注....24
1.3.4 小心....24

第 2 章
设置 RTCS

2.1 简介....25
2.2 支持的协议和政策....25
2.3 Freescale MQX RTOS 中包含的 RTCS....25
2.3.1 协议栈架构....28
2.4 设置 RTCS.... 28
2.5 定义 RTCS 协议.... 29
2.6 更改 RTCS 创建参数.... 30
2.7 创建 RTCS.... 30
2.8 更改 RTCS 运行参数.... 30
2.8.1 启用 IP 转发.... 30
2.8.2 略过 TCP 校验和.... 31
2.9 初始化设备接口....31
2.9.1 初始化以太网设备接口....31
2.9.1.1 获取以太网统计数据....31
2.9.2 初始化点对点设备接口....32
2.10 向 RTCS 中添加设备接口.... 32
2.10.1 从 RTCS 中删除设备接口....32

小节编号	标题	页

2.11 为设备接口绑定 IP 地址....................32

2.11.1 设备接口 IP 地址解绑....................32

2.12 添加网关....................33

2.12.1 添加默认网关....................33

2.12.2 为特定路由添加网关....................33

2.12.3 删除网关....................33

2.13 启用 RTCS 日志....................33

2.14 启动网络地址转换....................34

2.14.1 更改不活动超时....................34

2.14.2 指定端口范围....................35

2.14.3 禁用 NAT 应用层网关....................35

2.14.4 获取 NAT 统计数据....................35

2.14.5 支持的协议....................36

2.14.5.1 限制....................36

2.14.6 示例：设置 RTCS....................37

第 3 章
使用套接字

3.1 前言....................39

3.2 支持的协议....................39

3.3 套接字定义....................39

3.4 套接字选项....................40

3.5 数据报套接字和流套接字的比较....................40

3.6 数据报套接字....................41

3.6.1 无连接....................41

3.7 不可靠传输....................41

3.8 面向块....................41

3.9 流套接字....................41

3.10 基于连接....................41

3.11 可靠传输....................41

3.12 面向字符......42
3.13 创建和使用套接字......42
3.14 创建套接字......43
3.15 更改套接字选项......44
3.16 绑定套接字......44
3.17 使用数据报套接字......44
3.18 设置数据报套接字选项......44
3.19 传输数据报数据......45
3.19.1 缓冲......45
3.19.2 预先指定一个对等端......45
3.20 关闭数据报套接字......45
3.21 使用流套接字......46
3.22 更改流套接字选项......46
3.23 建立流套接字连接......46
3.23.1 被动建立流套接字连接......46
3.23.2 主动建立流套接字连接......46
3.24 获取流套接字名......47
3.25 发送流数据......47
3.26 接收流数据......47
3.27 缓冲数据......48
3.28 提高流数据的吞吐量......48
3.29 关闭流套接字......48
3.29.1 正常关闭......49
3.29.2 强制关闭......49
3.30 示例......50

第 4 章
点对点驱动程序

4.1 前言......53

小节编号	标题	页

4.2 PPP 及 PPP 驱动程序 53
4.2.1 LCP 配置选项 53
4.2.1.1 ACCM 54
4.2.1.2 ACFC 54
4.2.1.3 AP 55
4.2.1.4 MRU 55
4.2.1.5 PFC 55
4.2.2 配置 PPP 驱动程序 55
4.2.3 更改身份验证 56
4.2.3.1 PAP 57
4.2.3.2 CHAP 57
4.2.3.3 示例：设置 PAP 和 CHAP 身份验证 58
4.2.3.4 PAP — 客户端 58
4.2.3.5 CHAP — 客户端 58
4.2.3.6 PAP — 服务器端 59
4.2.3.7 CHAP — 服务器端 59
4.2.4 初始化 PPP 链路 60
4.2.4.1 使用多个 PPP 链路 60
4.2.5 获取 PPP 统计数据 60
4.2.6 示例：使用 PPP 驱动程序 60

第 5 章
RTCS 应用协议

5.1 前言 61
5.2 DHCP 客户端 61
5.2.1 示例：设置和使用 DHCP 客户端 62
5.3 DHCPv6 客户端 62
5.3.1 支持的功能 62
5.3.2 获取地址/其他配置 63
5.3.3 释放获取的地址 63

小节编号	标题	页

5.3.4 停止客户端 ... 63

5.4 DHCP 服务器 ... 63

5.4.1 示例：设置和修改 DHCP 服务器 ... 64

5.5 Echo 服务器 ... 64

5.6 Echo 客户端 ... 64

5.7 FTP 客户端 ... 65

5.8 FTP 服务器 ... 65

5.8.1 使用 FTP 客户端通信 ... 65

5.8.2 编译时配置 ... 66

5.8.3 基本用法 ... 67

5.9 LLMNR 服务器 ... 67

5.10 HTTP 服务器 ... 68

5.10.1 缓存控制 ... 68

5.10.2 支持的 MIME 类型 ... 69

5.10.3 别名 ... 69

5.10.4 编译时配置 ... 70

5.10.5 基本用法 ... 71

5.10.6 使用 CGI 回调 ... 71

5.10.7 使用服务器端包含（SSI）回调 ... 73

5.10.8 使用 CyaSSL 的安全 HTTP ... 73

5.10.9 分块传输编码 ... 73

5.10.10 HTTP 服务器存储器空间要求 ... 74

5.11 WebSocket 协议 ... 76

5.11.1 WebSocket API ... 77

5.11.2 以 HTTPSRV 插件的形式创建 WebSocket ... 77

5.11.3 通过 WebSocket 发送数据 ... 78

5.11.4 从 WebSocket 接收数据 ... 79

5.11.5 WebSocket 错误处理 ... 79

5.11.6 关闭 WebSocket 连接 ... 79

小节编号	标题	页

5.12 IPCFG — 高级网络接口管理 80
5.13 IWCFG — 高级无线网络接口管理 81
5.14 SMTP 客户端 81
5.14.1 发送邮件 81
5.14.2 示例应用 82
5.15 SNMP 代理 82
5.15.1 编译时配置 82
5.15.2 定义管理信息库（MIB） 83
5.15.3 向客户端应用发送一个陷阱消息 85
5.15.4 基本用法 86
5.16 简单网络时间协议（SNTP）客户端 86
5.17 Telnet 客户端 87
5.17.1 连接和断开 Telnet 服务器 87
5.18 Telnet 服务器 87
5.18.1 编译时配置 88
5.18.2 基本用法 88
5.19 TFTP 客户端 89
5.20 TFTP 服务器 90
5.20.1 编译时配置 90
5.20.2 基本用法 90
5.21 典型的 RTCS IP 数据包路径 91

第 6 章
重新编译

6.1 重新编译 RTCS 的原因 93
6.2 前言 93
6.3 Freescale MQX RTOS 中的 RTCS 编译工程 94
6.3.1 编译后处理 94
6.3.2 编译目标 94
6.4 重新编译 Freescale MQX RTCS 94

小节编号	标题	页

第 7 章
函数参考

7.1 函数列表格式....95

7.1.1 function_name()....95

7.2 accept()....96

7.3 ARP_stats()....99

7.4 bind()....100

7.5 closesocket()....102

7.6 connect()....104

7.7 DHCP_find_option()....106

7.8 DHCP_option_addr()....107

7.9 DHCP_option_addrlist()....108

7.10 DHCP_option_int16()....109

7.11 DHCP_option_int32()....110

7.12 DHCP_option_int8()....112

7.13 DHCP_option_string()....113

7.14 DHCP_option_variable()....114

7.15 DHCPCLN6_init()....115

7.16 DHCPCLN6_release()....116

7.17 DHCPCLN6_get_status()....117

7.18 DHCPCLNT_find_option()....118

7.19 DHCPCLNT_release()....118

7.20 DHCPSRV_init()....119

7.21 DHCPSRV_ippool_add()....121

7.22 DHCPSRV_set_config_flag_off()....122

7.23 DHCPSRV_set_config_flag_on()....123

7.24 ECHOCLN_connect....124

7.25 ECHOCLN_process....125

7.26 ECHOSRV_init()....126

小节编号	标题	页

7.27 ECHOSRV_release() 127

7.28 ENET_get_stats() 128

7.29 ENET_initialize() 129

7.30 FTP_close() 130

7.31 FTP_command_data() 131

7.32 FTP_open() 132

7.33 FTPSRV_init() 133

7.34 FTPSRV_release 134

7.35 getaddrinfo() 135

7.36 freeaddrinfo() 138

7.37 getnameinfo() 139

7.38 getpeername() 140

7.39 getsockname() 141

7.40 getsockopt() 142

7.41 HTTPSRV_init() 143

7.42 HTTPSRV_release() 144

7.43 HTTPSRV_cgi_write() 144

7.44 HTTPSRV_cgi_read() 145

7.45 HTTPSRV_ssi_write() 146

7.46 LLMNRSRV_init 147

7.47 LLMNRSRV_release 147

7.48 ICMP_stats() 148

7.49 IGMP_stats() 148

7.50 inet_pton() 149

7.51 inet_ntop() 150

7.52 IP_stats() 151

7.53 IPIF_stats() 152

7.54 ipcfg_init_device() 152

7.55 ipcfg_init_interface() 153

小节编号	标题	页

7.56 ipcfg_bind_boot() 155
7.57 ipcfg_bind_dhcp() 156
7.58 ipcfg_bind_dhcp_wait() 157
7.59 ipcfg_bind_staticip() 158
7.60 ipcfg_get_device_number() 159
7.61 ipcfg_add_interface() 160
7.62 ipcfg_get_ihandle() 160
7.63 ipcfg_get_mac() 161
7.64 ipcfg_get_state() 161
7.65 ipcfg_get_state_string() 162
7.66 ipcfg_get_desired_state() 163
7.67 ipcfg_get_link_active() 163
7.68 ipcfg_get_dns_ip() 164
7.69 ipcfg_add_dns_ip() 165
7.70 ipcfg_del_dns_ip() 165
7.71 ipcfg_get_ip() 166
7.72 ipcfg_get_tftp_serveraddress() 166
7.73 ipcfg_get_tftp_servername() 167
7.74 ipcfg_get_boot_filename() 167
7.75 ipcfg_poll_dhcp() 168
7.76 ipcfg_task_create() 169
7.77 ipcfg_task_destroy() 170
7.78 ipcfg_task_status() 171
7.79 ipcfg_task_poll() 171
7.80 ipcfg_unbind() 172
7.81 ipcfg6_bind_addr() 173
7.82 ipcfg6_unbind_addr() 174
7.83 ipcfg6_get_addr() 174
7.84 ipcfg6_get_dns_ip() 175

小节编号	标题	页

7.85 ipcfg6_add_dns_ip() ... 176
7.86 ipcfg6_del_dns_ip() ... 177
7.87 ipcfg6_get_scope_id() ... 178
7.88 iwcfg_set_essid() ... 178
7.89 iwcfg_get_essid() ... 179
7.90 iwcfg_commit() ... 180
7.91 iwcfg_set_mode() ... 181
7.92 iwcfg_get_mode() ... 181
7.93 iwcfg_set_wep_key() ... 182
7.94 iwcfg_get_wep_key() ... 183
7.95 iwcfg_set_passphrase() ... 183
7.96 iwcfg_get_passphrase() ... 184
7.97 iwcfg_set_sec_type() ... 185
7.98 iwcfg_get_sectype() ... 185
7.99 iwcfg_set_power() ... 186
7.100 iwcfg_set_scan() ... 186
7.101 listen() ... 187
7.102 MIB1213_init() ... 189
7.103 MIB_find_objectname() ... 189
7.104 MIB_set_objectname() ... 190
7.105 NAT_close() ... 191
7.106 NAT_init() ... 191
7.107 NAT_stats() ... 192
7.108 ping() ... 192
7.109 PPP_init() ... 192
7.110 PPP_release() ... 194
7.111 PPP_pause() ... 195
7.112 PPP_resume() ... 196
7.113 recv() ... 196

小节编号	标题	页

7.114 recvfrom() ... 198

7.115 RTCS_attachsock() ... 199

7.116 RTCS_create() ... 201

7.117 RTCS_detachsock() ... 201

7.118 RTCS_gate_add() ... 202

7.119 RTCS_gate_add_metric() ... 203

7.120 RTCS_gate_remove() ... 204

7.121 RTCS_gate_remove_metric() ... 205

7.122 RTCS_get_errno ... 205

7.123 RTCS_geterror() ... 206

7.124 RTCS_if_add() ... 207

7.125 RTCS_if_get_addr() ... 208

7.126 RTCS_if_get_handle () ... 209

7.127 RTCS_if_get_mtu() ... 210

7.128 RTCS_if_bind() ... 210

7.129 RTCS_if_bind_BOOTP() ... 211

7.130 RTCS_if_bind_DHCP() ... 212

7.131 RTCS_if_bind_DHCP_flagged() ... 214

7.132 RTCS_if_bind_DHCP_timed() ... 216

7.133 RTCS_if_bind_IPCP() ... 218

7.134 RTCS_if_rebind_DHCP() ... 220

7.135 RTCS_if_remove() ... 222

7.136 RTCS_if_get_link_status () ... 223

7.137 RTCS_if_unbind() ... 224

7.138 RTCS_if_get_dns_addr () ... 224

7.139 RTCS_if_add_dns_addr () ... 225

7.140 RTCS_if_del_dns_addr () ... 226

7.141 RTCS_ping() ... 227

7.142 RTCS_request_DHCP_inform() ... 228

小节编号	标题	页

7.143 RTCS_selectall()..........229
7.144 RTCS_selectset()..........231
7.145 RTCSLOG_disable()..........232
7.146 RTCSLOG_enable()..........233
7.147 RTCS6_if_bind_addr()..........234
7.148 RTCS6_if_unbind_addr()..........235
7.149 RTCS6_if_get_scope_id()..........236
7.150 RTCS6_if_get_prefix_list_entry()..........237
7.151 RTCS6_if_get_neighbor_cache_entry()..........238
7.152 RTCS6_if_get_addr()..........239
7.153 RTCS6_if_get_dns_addr ()..........240
7.154 RTCS6_if_add_dns_addr ()..........241
7.155 RTCS6_if_del_dns_addr ()..........242
7.156 RTCS6_if_is_disabled()..........243
7.157 select()..........244
7.158 RTCS_FD_SET..........248
7.159 RTCS_FD_CLR..........248
7.160 RTCS_FD_ZERO..........248
7.161 RTCS_FD_ISSET..........248
7.162 send()..........249
7.163 sendto()..........252
7.164 setsockopt()..........253
7.165 SOL_NAT_getsockopt..........270
7.166 SOL_NAT_setsockopt..........270
7.167 shutdownsocket()..........271
7.168 shutdown()..........273
7.169 SMTP_send_email..........274
7.170 SNMP_init()..........275
7.171 SNMP_trap_warmStart()..........276

小节编号	标题	页

7.172 SNMP_trap_coldStart().. 276
7.173 SNMP_trap_authenticationFailure().. 277
7.174 SNMP_trap_linkDown().. 277
7.175 SNMP_trap_myLinkDown()... 277
7.176 SNMP_trap_linkUp().. 278
7.177 SNMP_trap_userSpec()... 278
7.178 SNMPv2_trap_warmStart().. 278
7.179 SNMPv2_trap_coldStart().. 279
7.180 SNMPv2_trap_authenticationFailure()................................... 279
7.181 SNMPv2_trap_linkDown()... 279
7.182 SNMPv2_trap_linkUp()... 280
7.183 SNMPv2_trap_userSpec().. 280
7.184 SNTP_init().. 280
7.185 SNTP_oneshot()... 282
7.186 socket()... 282
7.187 TCP_stats().. 283
7.188 TFTPCLN_connect().. 284
7.189 TFTPCLN_get.. 285
7.190 TFTPCLN_put.. 286
7.191 TELNETSRV_init.. 286
7.192 TELNETSRV_release.. 287
7.193 TFTPSRV_init.. 288
7.194 TFTPSRV_release.. 289
7.195 UDP_stats().. 289
7.196 按服务列出函数.. 290

第 8 章
编译时选项

8.1 编译时选项.. 293
8.2 推荐设置.. 293

小节编号	标题	页

8.3 配置选项和默认设置......294

8.3.1 RTCSCFG_FD_SETSIZE...... 294

8.3.2 RTCSCFG_SOMAXCONN...... 294

8.3.3 RTCSCFG_ARP_CACHE_SIZE...... 294

8.3.4 RTCSCFG_IP_DISABLE_DIRECTED_BROADCAST...... 294

8.3.5 RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT......294

8.3.6 RTCSCFG_BOOTP_RETURN_YIADDR...... 294

8.3.7 RTCSCFG_DISCARD_SELF_BCASTS......295

8.3.8 RTCSCFG_UDP_ENABLE_LBOUND_MULTICAST......295

8.3.9 RTCSCFG_ENABLE_8021Q...... 295

8.3.10 RTCSCFG_LINKOPT_8023......295

8.3.11 RTCSCFG_DISCARD_SELF_BCASTS......295

8.3.12 RTCSCFG_ENABLE_ICMP...... 295

8.3.13 RTCSCFG_ENABLE_IGMP...... 296

8.3.14 RTCSCFG_ENABLE_NAT......296

8.3.15 RTCSCFG_ENABLE_IPIP......296

8.3.16 RTCSCFG_ENABLE_RIP......296

8.3.17 RTCSCFG_ENABLE_SNMP...... 296

8.3.18 RTCSCFG_ENABLE_SSL...... 296

8.3.19 RTCSCFG_ENABLE_IP_REASSEMBLY...... 296

8.3.20 RTCSCFG_ENABLE_LOOPBACK......296

8.3.21 RTCSCFG_ENABLE_UDP...... 297

8.3.22 RTCSCFG_ENABLE_TCP......297

8.3.23 RTCSCFG_ENABLE_STATS......297

8.3.24 RTCSCFG_ENABLE_GATEWAYS......297

8.3.25 RTCSCFG_ENABLE_VIRTUAL_ROUTES...... 297

8.3.26 RTCSCFG_USE_KISS_RNG...... 297

8.3.27 RTCSCFG_ENABLE_ARP_STATS...... 297

8.3.28 RTCSCFG_PCBS_INIT...... 297

小节编号	标题	页

8.3.29 RTCSCFG_LLMNRSRV_PORT.......298
8.3.30 RTCSCFG_LLMNRSRV_HOSTNAME_TTL.......298
8.3.31 RTCSCFG_PCBS_GROW.......298
8.3.32 RTCSCFG_PCBS_MAX.......298
8.3.33 RTCSCFG_MSGPOOL_INIT.......298
8.3.34 RTCSCFG_MSGPOOL_GROW.......298
8.3.35 RTCSCFG_MSGPOOL_MAX.......298
8.3.36 RTCSCFG_SOCKET_PART_INIT.......299
8.3.37 RTCSCFG_SOCKET_PART_GROW.......299
8.3.38 RTCSCFG_SOCKET_PART_MAX.......299
8.3.39 RTCSCFG_UDP_RX_BUFFER_SIZE.......299
8.3.40 RTCSCFG_ENABLE_UDP_STATS.......299
8.3.41 RTCSCFG_ENABLE_TCP_STATS.......299
8.3.42 RTCSCFG_TCP_MAX_CONNECTIONS.......299
8.3.43 RTCSCFG_TCP_MAX_HALF_OPEN.......299
8.3.44 RTCSCFG_ENABLE_RIP_STATS.......300
8.3.45 RTCSCFG_QUEUE_BASE.......300
8.3.46 RTCSCFG_STACK_SIZE.......300
8.3.47 RTCSCFG_LOG_PCB.......300
8.3.48 RTCSCFG_LOG_SOCKET_API.......300
8.3.49 RTCSCFG_ENABLE_IP4.......300
8.3.50 RTCSCFG_ENABLE_IP6.......300
8.3.51 RTCSCFG_ND6_NEIGHBOR_CACHE_SIZE.......301
8.3.52 RTCSCFG_ND6_PREFIX_LIST_SIZE.......301
8.3.53 RTCSCFG_ND6_ROUTER_LIST_SIZE.......301
8.3.54 RTCSCFG_IP6_IF_ADDRESSES_MAX.......301
8.3.55 RTCSCFG_IP6_IF_DNS_MAX.......301
8.3.56 RTCSCFG_IP6_REASSEMBLY.......301
8.3.57 RTCSCFG_IP6_LOOPBACK_MULTICAST.......302

小节编号 标题 页

8.3.58 RTCSCFG_ND6_RDNSS 302
8.3.59 RTCSCFG_ND6_RDNSS_LIST_SIZE 302
8.3.60 RTCSCFG_ND6_DAD_TRANSMITS 302
8.3.61 RTCSCFG_IP6_MULTICAST_MAX 302
8.3.62 RTCSCFG_IP6_MULTICAST_SOCKET_MAX 302
8.3.63 RTCSCFG_ENABLE_MLD 303
8.3.64 FTPCCFG_SMALL_FILE_PERFORMANCE_ENANCEMENT 303
8.3.65 FTPCCFG_BUFFER_SIZE 303
8.3.66 FTPCCFG_WINDOW_SIZE 303
8.3.67 RTCSCFG_ECHOSRV_DEBUG_MESSAGES 303
8.3.68 RTCSCFG_ECHOSRV_DEFAULT_BUFLEN 303
8.3.69 RTCSCFG_ECHOSRV_MAX_TCP_CLIENTS 303
8.3.70 RTCSCFG_ENABLE_SNMP_STATS 303
8.3.71 RTCSCFG_IPCFG_ENABLE_DNS 304
8.3.72 RTCSCFG_IPCFG_ENABLE_DHCP 304
8.3.73 RTCSCFG_IPCFG_ENABLE_BOOT 304
8.3.74 ENET 模块硬件加速选项 304
8.3.75 BSPCFG_ENET_HW_TX_IP_CHECKSUM_NEW 304
8.3.76 BSPCFG_ENET_HW_TX_PROTOCOL_CHECKSUM_NEW 304
8.3.77 BSPCFG_ENET_HW_RX_IP_CHECKSUM 304
8.3.78 BSPCFG_ENET_HW_RX_PROTOCOL_CHECKSUM_NEW 304
8.3.79 BSPCFG_ENET_HW_RX_MAC_ERR 305
8.3.80 RTCSCFG_ECHOCLN_DEFAULT_BUFLEN 305
8.3.81 RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT 305
8.3.82 RTCSCFG_ECHOCLN_DEBUG_MESSAGES 305

第 9 章
数据类型

9.1 RTCS 数据类型 307

小节编号	标题	页

9.2 按字母顺序排列的 RTCS 数据结构列表 307
9.2.1 addrinfo 307
9.2.2 ARP_STATS 309
9.2.3 BOOTP_DATA_STRUCT 310
9.2.4 DHCP_DATA_STRUCT 311
9.2.5 DHCPSRV_DATA_STRUCT 312
9.2.6 DHCPCLN6_STATUS 313
9.2.7 DHCPCLN6_PARAM_STRUCT 313
9.2.8 ECHOSRV_PARAM_STRUCT 313
9.2.9 ENET_STATS 314
9.2.10 FTPSRV_AUTH_STRUCT 316
9.2.11 FTPSRV_PARAM_STRUCT 317
9.2.12 HTTPSRV_PARAM_STRUCT 318
9.2.13 HTTPSRV_AUTH_USER_STRUCT 320
9.2.14 HTTPSRV_AUTH_REALM_STRUCT 321
9.2.15 HTTPSRV_CGI_REQ_STRUCT 321
9.2.16 HTTPSRV_CGI_RES_STRUCT 323
9.2.17 HTTPSRV_SSI_PARAM_STRUCT 323
9.2.18 HTTPSRV_SSI_LINK_STRUCT 324
9.2.19 HTTPSRV_CGI_LINK_STRUCT 324
9.2.20 HTTPSRV_ALIAS 325
9.2.21 HTTPSRV_PLUGIN_STRUCT 325
9.2.22 HTTPSRV_PLUGIN_LINK_STRUCT 325
9.2.23 HTTPSRV_SSL_STRUCT 326
9.2.24 PING_PARAM_STRUCT 326
9.2.25 ICMP_STATS 326
9.2.25.1 ST_RX_TOTAL 327
9.2.26 IGMP_STATS 330
9.2.27 in_addr 331

小节编号	标题	页

9.2.28 in6_addr ... 331
9.2.29 ip_mreq ... 332
9.2.30 ipv6_mreq ... 332
9.2.31 IP_STATS ... 333
9.2.32 IPCFG_IP_ADDRESS_DATA ... 335
9.2.33 IPCP_DATA_STRUCT ... 336
9.2.34 IPIF_STATS ... 338
9.2.35 LLMNRSRV_PARAM_STRUCT ... 339
9.2.36 LLMNRSRV_HOST_NAME_STRUCT ... 340
9.2.37 nat_ports ... 341
9.2.38 NAT_STATS ... 341
9.2.39 nat_timeouts ... 342
9.2.40 PPP_PARAM_STRUCT ... 342
9.2.41 PPP_SECRET ... 343
9.2.42 RTCS_ERROR_STRUCT ... 344
9.2.43 RTCS_IF_STRUCT ... 345
9.2.44 rtcs_fd_set ... 346
9.2.45 RTCS_protocol_table ... 347
9.2.46 RTCS_SSL_PARAMS_STRUCT ... 348
9.2.47 RTCS_TASK ... 348
9.2.48 RTCS6_IF_ADDR_INFO ... 349
9.2.49 ssRTCS6_IF_PREFIX_LIST_ENTRY ... 349
9.2.50 RTCS6_IF_NEIGHBOR_CACHE_ENTRY ... 350
9.2.51 rtcs6_if_addr_type ... 350
9.2.52 RTCSMIB_VALUE ... 350
9.2.53 SMTP_EMAIL_ENVELOPE 结构 ... 351
9.2.54 SMTP_PARAM_STRUCT 结构 ... 351
9.2.55 sockaddr_in ... 352
9.2.56 sockaddr_in6 ... 353

小节编号	标题	页

9.2.57 sockaddr ... 353
9.2.58 TCP_STATS ... 354
9.2.59 TELNETCLN_CALLBACK ... 358
9.2.60 TELNETCLN_CALLBACKS_STRUCT ... 358
9.2.61 TELNETCLN_PARAM_STRUCT ... 359
9.2.62 TELNETSRV_PARAM_STRUCT ... 359
9.2.63 TFTPCLN_PARAM_STRUCT ... 361
9.2.64 TELNETCLN_DATA_CALLBACK ... 361
9.2.65 TELNETCLN_ERROR_CALLBACK ... 362
9.2.66 TFTPSRV_PARAM_STRUCT ... 362
9.2.67 UDP_STATS ... 363
9.2.68 WS_DATA_STRUCT ... 364
9.2.69 WS_PLUGIN_STRUCT ... 365
9.2.70 WS_USER_CONTEXT_STRUCT ... 366

第 1 章
前言

1.1 关于本手册

本手册为 MQX™ RTCS™ Embedded TCP/IP 协议栈的使用参考手册，该协议栈是 Freescale MQX 实时操作系统发行版的组成部分。

本文档针对有经验的软件开发者而撰写，读者应具备 C 和 C++语言及其目标处理器的相关工作知识。

1.2 如何获取更多信息

- Freescale MQX RTOS 发布时随附的版本记录文档中提供了本用户指南出版时可用的信息。
- 《*MQX RTOS* 用户指南》中描述了如何创建使用 MQX RTOS 的嵌入式应用。
- 《*MQX RTOS* 参考手册》中描述了 MQX RTOS API 的原型。

1.3 约定

本节将解释本手册中使用的的术语和其他约定。

1.3.1 产品名称

- RTCS：本手册中使用 RTCS 作为 MQX RTCS 全功能 TCP/IP 协议栈的简称。
- MQX RTOS：MQX RTOS 为 MQX 实时操作系统的简称。

1.3.2 提示

提示指出有用信息。

提示

如果您的 CD-ROM 驱动器使用其他驱动器盘符，请相应地替换本命令中的驱动器盘符。

1.3.3 附注

附注指出重要信息。

注

非严格信号量没有优先级继承。

1.3.4 小心

"小心"涉及到可能产生意外效果或不利影响、或者可能给文件或硬件带来危险的命令或步骤。

警告

如果您修改 MQX RTOS 数据类型，某些工具可能无法正常运行。

第 2 章
设置 RTCS

2.1 简介

本章描述了如何配置、创建和设置 RTCS，以准备使用套接字。

相关信息	参见
本章中提到的数据类型	第 8 章"数据类型"
PPP 驱动程序	第 4 章"点对点驱动程序"
协议	附录 A"协议和政策"
本章中提到的函数的原型。	第 7 章"函数参考"
套接字	第 3 章"使用套接字"

2.2 支持的协议和政策

图 2-1 显示了本手册中讨论的协议与政策。关于协议的详细信息，参见上表及 附录 A"协议与约定" 。

2.3 Freescale MQX RTOS 中包含的 RTCS

Freescale MQX RTOS 发行版中包含的 RTCS 协议栈基于 ARC RTCS 2.97 版。关于支持的 RTCS 新特性，请参见 Freescale MQX RTOS 随附的版本记录文档 。

Freescale MQX RTOS 发行版中 RTCS 的重大变更:

- 目前，RTCS 在 Freescale MQX RTOS 程序包内发布。从 3.0 版本开始，RTCS 采用 Freescale MQX RTOS 发行版的版本编号方案。
- RTCS 的编译过程与编译时配置遵循的原则与其它 MQX RTOS 内核库相同。第 6 章"重新编译" 。

- RTCS Shell 和所有 Shell 函数均从 RTCS 库中转移到 Freescale MQX 发行版内一个单独的库中。
- Freescale MQX RTOS 发行版中添加了一个新的 HTTP 服务器功能。

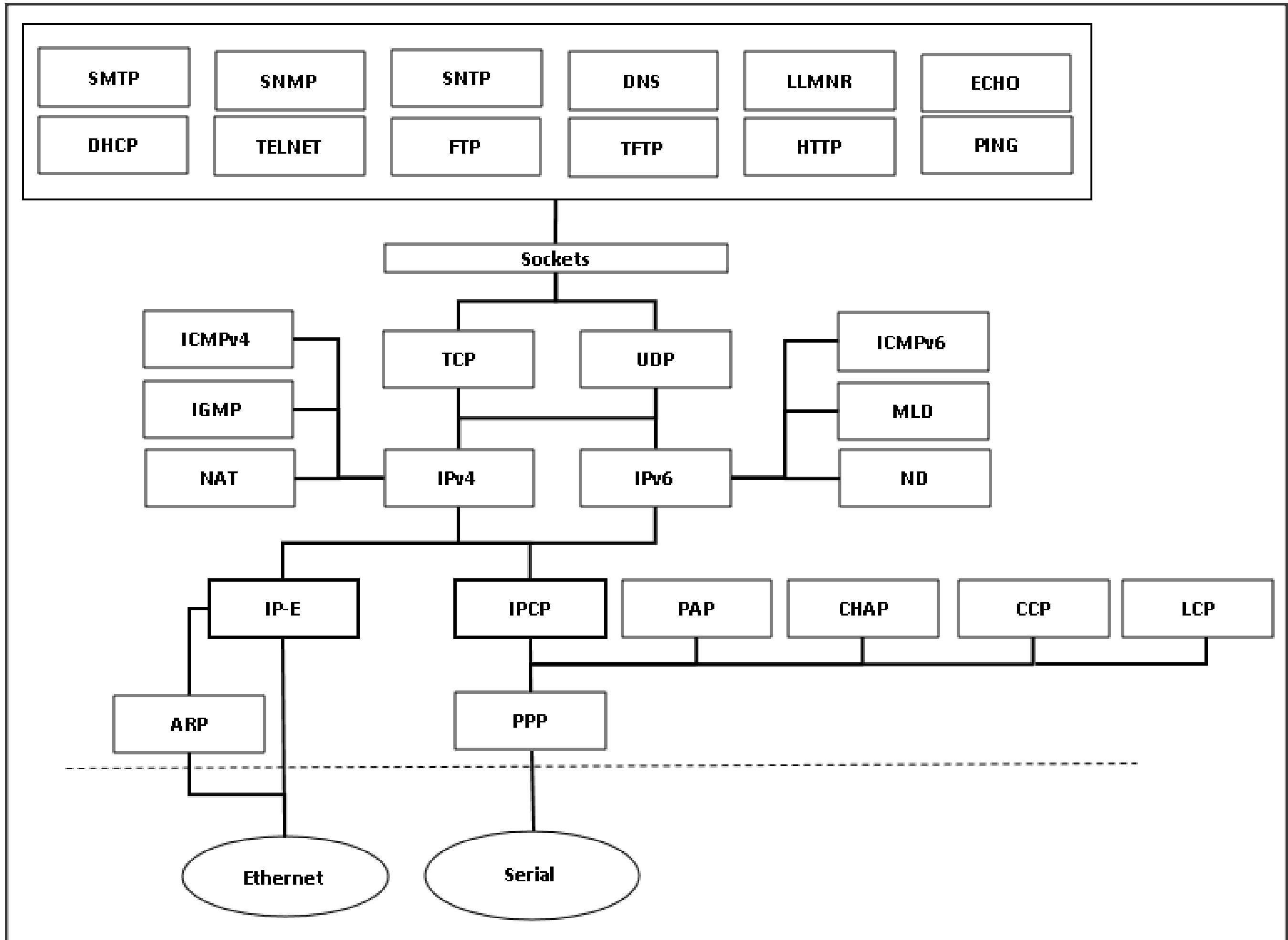

图 2-1. 协议

表 2-1. RTCS 特性

协议或政策	说明	RFC
ARP	以太网地址解析协议	826
BootP	引导程序协议	951, 1542
CCP	压缩控制协议（用于 PPP）	1692
CHAP	质询握手认证协议（用于 PPP）	1334
CIDR	无类别域间路由	1519
DHCP	动态主机配置协议	2131
DHCP 选项	DHCP 选项与 BootP 供应商扩展	2132
DNS	域名：实现与规范	1035

下一页继续介绍此表...

表 2-1. RTCS 特性 (继续)

协议或政策	说明	RFC
Echo	Echo 协议	862
Ethernet		(IEEE 802.3)
FTP	文件传输协议	959
HDLC	高级数据链路控制协议	(ISO 3309)
HTTP	超文本传输协议	2068
ICMP	互联网控制消息协议	792
IGMP	互联网组管理协议	1112
IP	互联网协议	791, 919, 922
	在有子网的情况下广播互联网数据报	922
	互联网标准子网划分过程	950
IPCP	IP 控制协议（用于 PPP）	1332
IP-E	以太网 IP 数据报传输标准	894
IPIP	IP 隧道中的 IP	1853
LCP	链路控制协议（用于 PPP）	1661, 1570
MD5	RSA Data Security Inc.的 MD5 信息提取算法	1321
MIB	管理信息库（SNMPv2 的一部分）	1902, 1907
NAT	网络地址转换	
	传统 IP 网络地址转换器（传统 NAT）	3022
	IP 网络地址转换器 (NAT)的术语和注意事项	2663
PAP	密码认证协议（用于 PPP）	1334
ping	通过 ICMP Echo 消息实现	792
PPP	点对点协议	1661
PPP（类 HDLC 帧）	类 HDLC 帧中的 PPP	1662
PPP LCP 扩展		1570
Quote	每日引语（QOTD）协议	865
Reqs	互联网主机要求:	
	通信层	1122
	应用与支持协议	1123
	IPv4 路由要求	1812
RIP	路由信息协议	2453
SMI	管理信息结构	1155
SNMPv1 MIB	SNMPv1 管理信息库	1213
SNMPv2	SNMP 版本 2	1902 — 1907
SNMPv2 MIB	SNMPv2 管理信息库	1902, 1907
SNTP	简单网络时间协议	2030
TCP	传输控制协议	793

下一页继续介绍此表...

表 2-1. RTCS 特性 (继续)

协议或政策	说明	RFC
Telnet	Telnet 协议规范	854
TFTP	简单文件传输协议	1350
UDP	用户数据报协议	768

2.3.1 协议栈架构

图 2-2 显示了 RTCS 协议栈的架构及 RTCS 与上下各层之间的通信方式。

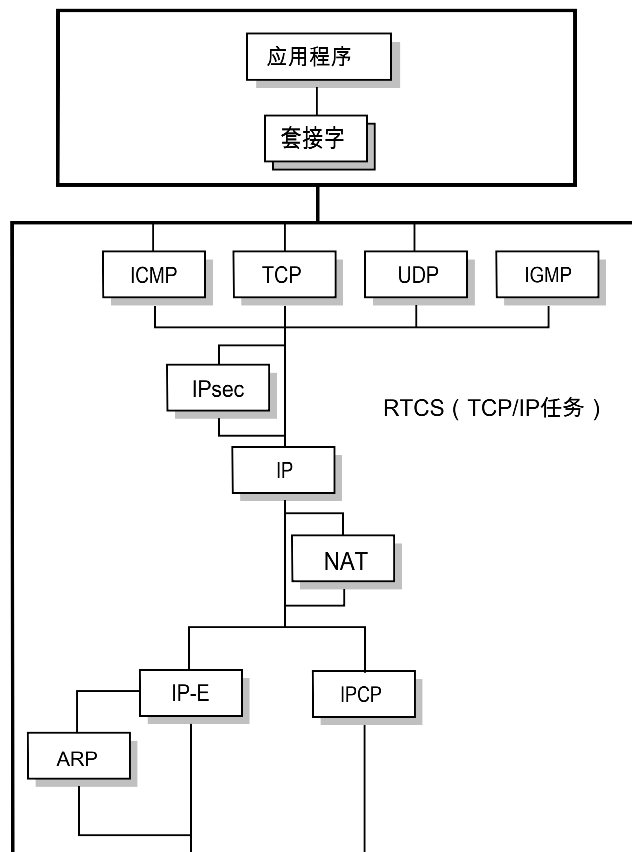

图 2-2. 协议栈架构

2.4 设置 RTCS

应用应遵循一系列常规步骤设置 RTCS。这些步骤如图 2-3 中所示，在后续章节中有详细说明。

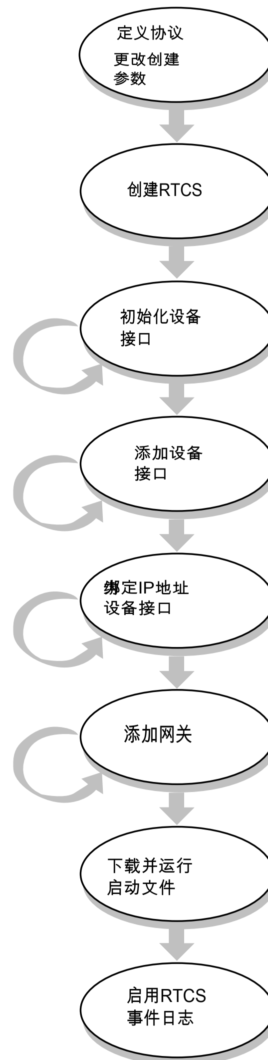

图 2-3. RTCS 的设置步骤

2.5 定义 RTCS 协议

应用在创建 RTCS 时通过一个协议表确定将启动哪些协议，以及各协议启动的顺序。关于可用协议的列表，请参见第 8 章"数据类型" 中的 8.2.36 节 "RTCS_protocol_table" 。您可以使用其中提供的指令来添加或删除协议，以生成自己的协议表。

2.6 更改 RTCS 创建参数

在应用创建 RTCS 时，RTCS 会使用几种全局变量。所有变量都具有默认值，对于大多数变量来说，如果想要更改默认值，应用都必须在创建 RTCS 或调用 RTCS_create() 之前予以更改。

更改目标：	原默认值：	更改此创建变量：
RTCS 任务的优先级如果 RTCS 任务的优先级太低，RTCS 可能会丢失收到的数据包或违背协议的时序规范。	6	_RTCSTASK_priority
RTCS 任务（TCP/IP）协议栈大小	3000	_RTCSTASK_stacksize
RTCS 使用的数据包控制块（PCB）的最大个数	10	_RTCSPCB_max
RTCS 从中分配内存的池。如果为 0，将使用系统池。如果要使用其他的池，则必须提供内存池的编号。 示例： `_RTCS_mem_pool = _mem_create_pool(ADR, SIZE)`	0	_RTCS_mem_pool

2.7 创建 RTCS

若要创建 RTCS，调用 RTCS_create() 可分配 RTCS 需要的资源并创建 RTCS 任务。

2.8 更改 RTCS 运行参数

在应用创建 RTCS 后，RTCS 会使用一些全局变量。所有变量都具有默认值，对于大多数变量来说，如果想要更改默认值，应用都必须在创建 RTCS 或调用 RTCS_create() 之后予以更改。

若要：	将此变量更改为 **TRUE**：
启用 IP 转发和网络地址转换 （NAT 或 IPShield 所需）。	_IP_forward
对传入数据包不验证 TCP 校验和。	_TCP_bypass_rx
对传出数据包不生成 TCP 校验和。	_TCP_bypass_tx

2.8.1 启用 IP 转发

此参数提供了在 NAT 或 IPShield 所需网络接口之间路由数据包的功能。

2.8.2 略过 TCP 校验和

如果数据传输性能较为重要，您可能希望在隔离的网络中略过 TCP 校验和的生成和验证。

对于传入数据包，如果您略过 TCP 校验和验证，RTCS 将不检测数据流中出现的错误。但出现这些错误的概率较低，因为最底层还包含一个用来检测数据流错误的校验和。

2.9 初始化设备接口

RTCS 支持任何根据已发布标准编写的驱动程序，例如，PPP、IPCP 和以太网上的 PPP。

由于 RTCS 独立于任何设备，对应用正在使用（或计划用于）连接网络的设备，它没有任何内建知识。因此， 应用必须:

- 对每个设备的每个接口进行初始化。
- 设置每个接口的状态，使其可以发送和接收网络通信流。
- 将每个受支持的设备动态添加到 RTCS。

当应用初始化一个设备接口时，初始化函数将返回该接口的一个句柄。随后，应用通过引用这个设备句柄将接口添加到 RTCS，并为该接口绑定 IP 地址。

2.9.1 初始化以太网设备接口

在应用使用一个以太网设备接口之前，必须通过调用 **ENET_initialize()**函数初始化设备驱动程序接口。此函数执行下列任务:

- 初始化以太网硬件，使其准备好发送和接收以太网数据包。
- 安装以太网驱动程序的中断服务例程（ISR）。
- 设定发送与接收缓冲区，通常表示以太网设备自身的缓冲区。
- 分配和初始化以太网设备句柄，供应用从以太网驱动程序 API（**ENET_get_stats()**）和 RTCS API 中调用函数时使用。

2.9.1.1 获取以太网统计数据

若要获取关于以太网接口的统计数据，请调用 **ENET_get_stats()**，将设备句柄传递给接口。

2.9.2 初始化点对点设备接口

点对点设备使用 PPP 协议及以太网上的 PPP 协议。关于初始化点对点设备接口的详细内容，请参见第 4 章"点对点驱动程序"。

2.10 向 RTCS 中添加设备接口

应用对设备接口进行初始化后，通过设备句柄调用 **RTCS_if_add()**，以向 RTCS 中添加各接口。

2.10.1 从 RTCS 中删除设备接口

若要从 RTCS 中删除一个设备接口，需使用设备句柄调用 **RTCS_if_remove()**。

2.11 为设备接口绑定 IP 地址

应用向 RTCS 添加设备接口后，将为每个接口绑定一个或多个 IP 地址。

一个应用可通过多种方式为设备接口绑定 IP 地址。

若要:		调用:
绑定应用指定的一个 IP 地址。		**RTCS_if_bind()**
绑定通过下列方式获取的一个 IP 地址:		
	BootP	**RTCS_if_bind_BOOTP()**
	DHCP	**RTCS_if_bind_DHCP()**
	IPCP（适用于 PPP 的唯一方法）	**RTCS_if_bind_IPCP()**

2.11.1 设备接口 IP 地址解绑

若要从一个设备接口上解除绑定的 IP 地址，请调用 **RTCS_if_unbind()**。

2.12 添加网关

RTCS 通过网关与远程子网通信。应用一般在设置 RTCS 时添加网关，但也可在任何时候添加网关。若要添加一个网关，请调用 **RTCS_gate_add()**，并提供网关 IP 地址和网络掩码。

2.12.1 添加默认网关

若要添加默认网关，请调用：

```
RTCS_gate_add(ip_address, 0, 0)
```

2.12.2 为特定路由添加网关

若要添加一个地址为 *ip_address* 的网关，以连接到子网 192.168.1.0/24，请调用：

```
RTCS_gate_add(ip_address, 0xC0A80100, 0xFFFFFF00)
```

2.12.3 删除网关

若要删除一个网关，请调用 **RTCS_gate_remove()**。

2.13 启用 RTCS 日志

您可以在 MQX RTOS 内核日志中启用 RTCS 事件日志。性能分析工具使用内核日志数据来分析应用工作情况和资源使用情况。

启用 RTCS 日志前，您必须在 RTCSCFG_LOGGING 定义为 1 的条件下对 MQX RTOS（RTCS 库）进行编译。关于内核日志的编译参数，请参见《*MQX RTOS* 用户指南》。

用户必须在应用中创建内核日志并启用 RTCS 日志(KLOG_RTCS_FUNCTIONS)。关于内核日志的详细描述，请参见《*MQX RTOS* 用户指南》。通过调用 **RTCSLOG_enable()**并提供所需的事件掩码来启用 RTCS 事件日志。调用 **RTCSLOG_disable()**可禁用 RTCS 事件日志。

2.14 启动网络地址转换

NAT 允许站点使用专用地址来初始化对外网主机的单向出站访问。支持网络地址端口转换。

当 NAT 启用后，NAT 路由器（本例中为 RTCS）将保留一个外部可路由 IP 地址块，以代表边界路由器后各主机不可路由的专用地址。通过少量的可路由地址即可支持大型主机池共享 NAT 连接。

当一个数据包离开专用网络时，边界路由器将源 IP 地址转换为保留池中的地址，并将源传输标识（TCP/UDP 端口或 ICMP 查询 ID）转换为一个选定的随机数。收到响应时，边界路由器将此随机的 NAT 流标识进行逆向转换，映射出原始发送方的 IP 地址以及该主机在专用网络中的传输标识。

路由器将所有入站数据包的目标地址和相关字段转换为在专用网络中的主机地址、传输 ID 和相关字段。

若要启动网络地址转换，应用程序需要调用 NAT_init() 并且使用专用网络地址及子网掩码作为参数 。进行网络地址转换之前，全局 RTCS 运行参数 *_IP_forward* 的值必须为 TRUE。

在初始化时需要为内部配置结构分配空间。该配置结构包括:

- 地址空间分区。
- 维护状态信息。
- 指向应用层网关列表。
- 对不活动连接进行连接超时设置。
- 标识专用网络上通过 NAT 管理的端口和 ICMP 查询 ID。

2.14.1 更改不活动超时

NAT 启动后，通过 RTCS 事件队列来监视专用主机和公共主机之间的会话。事件计时器用来确定会话结束的时间。*nat.h* 头文件中定义了在终止一个不活动的 UDP 或 TCP 会话前需等待的时间，可通过 SOL_NAT_setsockopt() 函数进行动态配置。

SOL_NAT_setsockopt() 被调用时，应用向其传递 NAT 超时结构的地址 *nat_timeouts*。该结构针对下列情况提供三种不活动超时值:

- TCP 会话 — 默认超时为 15 分钟。

- UDP 或 ICMP 会话 — 默认超时为五分钟。
- TCP 会话（其中 FIN 或 RST 位已置位）— 默认超时为两分钟。

应用每次提供 *nat_timeouts* 结构时都会重写这三个值。为避免更改已有的超时值，应用必须针对特定的超时提供一个零值。

2.14.2 指定端口范围

在一个会话中，NAT 使用的所有端口位于 nat.h 头文件中定义的指定范围内。这些端口的范围可通过 SOL_NAT_setsockopt() 函数进行动态更改，该函数需要接收一个 NAT 端口结构 nat_ports。此结构中含有 NAT 使用的端口号（TCP、UDP 和 ICMP ID）的上限和下限。在默认情况下，最小端口号为 10000，最大端口号为 20000。

每当应用给出一个 nat_ports 结构时，都会重写最小端口号和最大端口号。为避免更改现有的端口号，应用必须先将最小或最大值设为 0。

应用不能使用预留端口，ICMP 查询不能将这些端口作为序号使用。当会话结束时，NAT 会自动执行地址解绑和清理。

2.14.3 禁用 NAT 应用层网关

NAT 启动后，活动 TFTP ALG 和 FTP ALG 常驻在 NAT 设备上。如果执行应用特定的载荷监视和变更时无需这些网关，则可在编译时通过重定义 NAT_alg_table 表将其禁用。此表根据源端口或目标端口的 TFTP 和 FTP 修正和确认数据。

NAT_alg_table 表在 natalg.c 中定义。其中含有一个指向 ALG 的函数指针数组。应用只能使用表中的 ALG。若您从表中删除一个 ALG，RTCS 则无法使相关代码链接到您的应用。

默认情况下，此表按如下方式定义：

```
NAT_ALG NAT_alg_table[] = {
    NAT_ALG_TFTP,
    NAT_ALG_FTP,
    NAT_ALG_ENDLIST
};
```

若要禁用 TFTP、FTP 和 NAT 负载监视和变更，在编译时按如下方式对此表重定义：

```
NAT_ALG NAT_alg_table[] = {
   NAT_ALG_ENDLIST
};
```

2.14.4 获取 NAT 统计数据

统计数据通过在 nat.h 中定义的 NAT_STATS 结构提供。应用程序调用 NAT_stats() 以获取 NAT 的统计数据。

2.14.5 支持的协议

Freescale MQX RTOS 实现的 NAT 支持采用如下协议进行通信:

- 数据中不包含端口与地址信息的 TCP 会话和 UDP 会话
- ICMP
- HTTP
- Telnet
- Echo
- TFTP 与 FTP

无论采用何种协议来传输数据包, NAT 对专用网络内部主机之间传递的数据包没有影响。关于 NAT 的详细信息，参见 附录 A"协议与约定" 。

2.14.5.1 限制

Freescale MQX RTOS 实现的 NAT 不支持:

- IGMP 与 IP 多播模式
- 零散的 TCP 与 UDP 数据包
- IKE 与 IPsec
- SNMP
- 专用主机的公共 DNS 查询
- H.323
- 点对点连接。只有专用主机可以发起通往公共主机的连接。

此外，对于单一的专用网络，Freescale MQX RTOS 实现的 NAT 只能工作在边界路由器上。

表 2-2. 一览表：设置函数

NAT_close	停止网络地址转换。
NAT_init	启动网络地址转换。
RTCS_create	创建 RTCS。
RTCS_gate_add	向 RTCS 添加一个网关。
RTCS_gate_remove	从 RTCS 中删除一个网关。
RTCS_if_add	向 RTCS 添加一个设备接口。
RTCS_if_bind	将一个 IP 地址绑定到一个设备接口。
RTCS_if_bind_BOOTP	使用 BootP 获得一个 IP 地址以绑定到设备接口。
RTCS_if_bind_DHCP	使用 DHCP 获得一个 IP 地址以绑定到设备接口。
RTCS_if_bind_IPCP	将一个 IP 地址绑定到一个 PPP 链路。
RTCS_if_remove	从 RTCS 中删除一个设备接口。
RTCS_if_unbind	从一个设备接口上解除绑定的 IP 地址。
RTCSLOG_enable	启用 RTCS 事件日志。
RTCSLOG_disable	禁用 RTCS 事件日志。
SOL_NAT_setsockopt()	设定 NAT 选项。

2.14.6 示例：设置 RTCS

在一个以太网设备上设置 RTCS:

```
_rtcs_if_handle   ihandle;
uint32_t           error;
/* For Ethernet driver: */
_enet_handle   ehandle;
/* For PPP Driver: */
FILE_PTR           pfile;
/* Change the priority: */
_RTCSTASK_priority = 7;
error = RTCS_create();
if (error) {
  printf("\nFailed to create RTCS, error = %X", error);
  return;
}
/* Enable IP forwarding: */
   _IP_forward = TRUE;

/* Set up the Ethernet driver: */
error = ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
if (error) {
    printf("\nFailed to initialize Ethernet driver: %s",
           ENET_strerror(error));
    return;
}
error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
  printf("\nFailed to add interface for Ethernet, error = %x",
         error);
    return;
}
error = RTCS_if_bind(ihandle, enet_ipaddr, enet_ipmask);
if (error) {
   printf("\nFailed to bind interface for Ethernet, error = %x",
```

```
            error);
   return;
}
printf("\nEthernet device %d bound to %X",
      ENET_DEVICE, enet_ipaddr);

/* Install a default gateway: */
RTCS_gate_add(GATE_ADDR, INADDR_ANY, INADDR_ANY);
```

第 3 章
使用套接字

3.1 前言

本章介绍如何使用 RTCS 及其套接字。应用程序设置 RTCS 后，将会使用套接字接口通过 TCP/IP 网络与其他应用或服务器通信。

相关信息	参见
本章中提到的数据类型	第 8 章“数据类型”
MQX RTOS	*MQX RTOS* 用户指南 *MQX RTOS* 参考手册
协议	附录 A“协议和政策”
本章中提到的函数的原型。	第 7 章“函数参考”
设置 RTCS	第 2 章 “设置 RTCS”

3.2 支持的协议

RTCS 套接字为以下协议提供接口:

- TCP
- UDP

3.3 套接字定义

套接字是识别端点的抽象，包括:

- 套接字类型，包括:

    - 数据报类型（使用 UDP）
    - 流类型（使用 TCP）
- 套接字地址，标识方式为：
    - 端口号
    - IP 地址

一个套接字可能具有一个远程端点。

3.4 套接字选项

每个套接字具有一些套接字选项，用来定义套接字的特征，例如：

- 校验和计算
- 以太网帧特征
- IGMP 成员属性
- 无阻塞（无等待选项）
- 压栈操作
- 发送和接收缓冲区的大小
- 超时

3.5 数据报套接字和流套接字的比较

表 3-1 概括了数据报套接字和流套接字的不同点。

表 3-1. 数据报套接字和流套接字

套接字类型	数据报套接字	流套接字
协议	UDP	TCP
基于连接	否	有
可靠传输	否	有
传输模式	面向块	面向字符

3.6 数据报套接字

3.6.1 无连接

数据报套接字是无连接的，因为应用使用套接字之前无需先建立连接。因此，应用在每次数据传输时都要指定目标地址和目标端口号。如果需要，应用可以为数据报套接字预先指定一个远程端点。

3.7 不可靠传输

数据报套接字用于基于数据报的数据传输，不对传输进行确认回复。因为不保证数据交付成功， 所以在必要时应用要负责确认数据。

3.8 面向块

数据报套接字是面向块的套接字，当应用序发送一个数据块时，数据的各个字节保持在一起。例如，若应用写一个 100 字节的数据块，RTCS 通过单个数据包将这些数据发送至目标地址，而目标地址将收到 100 字节的数据。

3.9 流套接字

3.10 基于连接

对于一个流套接字连接，通过两端点中每个端点的地址和端口号组合可以唯一地确定这个连接。例如，与 Telnet 服务器的一个连接使用本地 IP 地址+本地端口号，以及服务器 IP 地址+端口号 23。

3.11 可靠传输

流套接字可实现可靠的、端到端数据传输。若要使用流套接字，客户端要与一个对等端建立一条连接，然后传输数据，最后关闭连接。除非物理连接断开，否则 RTCS 能保证所有发送的数据都按序接收。

3.12 面向字符

流套接字是面向字符的套接字。这意味着 RTCS 从一个协议栈向另一个协议栈发送数据时，可能会将数据按字节进行拆分或者合并。例如，一个流套接字的应用可能会执行两个连续的写操作，每次写 100 个字节，而 RTCS 通过单个数据包将数据发往目标端口。然后目标端口可能会通过 4 次连续读操作来接收这些数据，每次读取 50 个字节。

3.13 创建和使用套接字

应用在创建和使用套接字时遵循以下通用步骤。下面的示意图中对这些步骤进行了概述，后续章节中还将对此进行详细说明。

- 通过调用 **socket()**创建一个新的套接字，指明该套接字是数据报套接字还是流套接字。
- 通过调用 **bind()**将套接字与一个本地地址绑定。
- 如果该套接字为流套接字，选用下列任一方式为其分配一个远程 IP 地址:
    - 调用 **connect()**。
    - 调用 **listen()**，随后调用 **accept()**。
- 发送数据，实现方式：数据报套接字调用 **sendto()**，流套接字调用 **send()**。
- 接收数据，实现方式：数据报套接字调用 **recvfrom()**，流套接字调用 **recv()**。
- 数据传输结束后，可调用 **shutdown()**销毁套接字。

创建和使用数据报套接字的过程如图 3-1 所示。

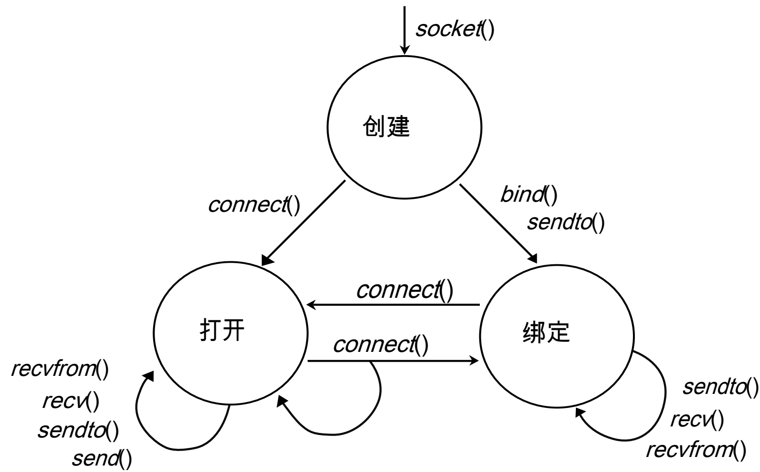

图 3-1. 创建和使用数据报套接字（UDP）

创建和使用流套接字的过程如图 3-2 所示。

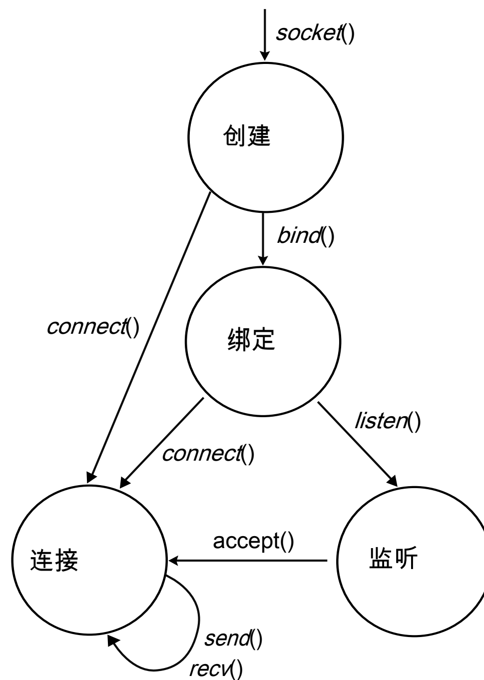

图 3-2. 创建和使用流套接字（TCP）

3.14 创建套接字

若要创建一个套接字，应用调用 **socket()**并指明该套接字是数据报套接字还是流套接字。该函数返回一个套接字句柄，应用可以通过其访问此套接字。

3.15 更改套接字选项

RTCS 在创建套接字时将所有套接字选项设置为默认值。对于某些特定选项，应用只能在绑定套接字之前更改其值。而对于其他选项,应用可以在任何时候更改其值。

关于所有套接字选项及默认值，参见第 7 章"函数参考" 中 **setsockopt()** 的列表。

3.16 绑定套接字

创建了一个套接字并且更改或设置套接字选项（根据需要）后，应用必须通过调用 **bind()**将该套接字与一个本地端口号绑定。该函数通过本地 IP 地址和端口号定义本地套接字的终结点。

本地端口号可指定为任何数字，但若指定为零，RTCS 将会选择一个未使用的端口号。若要确定 RTCS 所选的端口号，调用 **getsockopt()**。

绑定套接字之后，应用对套接字的使用方式取决于此套接字是数据报套接字还是流套接字。

3.17 使用数据报套接字

3.18 设置数据报套接字选项

在默认情况下，RTCS 使用 IGMP，并且套接字不属于任何分组。应用可以更改套接字的如下选项:

- IGMP 加入多播组
- IGMP 退出多播组
- 发送无等待
- 略过校验和

关于选项的更多详情，参见第 7 章"函数参考" 中 **setsockopt()** 的列表。

若要更改默认方式以使 RTCS 不使用 IGMP，请参见 2.5 节 "定义 RTCS 协议" 。

3.19 传输数据报数据

应用通过调用 **sendto()**或 **send()**和 **recvfrom()**或 **recv()**来传输数据。在每个调用过程中，RTCS 发送或接收一个 UDP 数据报，该数据报最多可包含 65,507 字节的数据。如果应用指定了更多的数据，函数将返回错误。

默认情况下，send()和 sendto()在数据被复制到套接字层并计划发送（无阻塞方式）时返回。

函数 **recv()**和 **recvfrom()**在套接字端口接收数据包时返回，或立即返回（如果排队的数据包已经到达端口）。接收缓冲区的大小应至少匹配应用预期接收的最大数据报。如果数据包超过接收缓冲区的限度，RTCS 将截断该数据包并丢弃截下的数据。

3.19.1 缓冲

在默认情况下，**send()**和 **sendto()**用来缓冲发送的数据。可通过 OPT_SEND_NOWAIT 套接字选项或 RTCS_MSG_BLOCK 发送标志更改此方式。

对于接收的数据，RTCS 将数据逐包匹配应用对 **recv()**或 **recvfrom()**的调用。如果在数据包到达时，**recv()**或 **recvfrom()**的某个调用并未等待数据，则 RTCS 将此包排入队列。

3.19.2 预先指定一个对等端

应用可以通过调用 **connect()**预先指定一个对等端。预先指定具有如下作用：

- 可通过 **send()**函数向 **connect()**被调用时指定的对等端发送一个数据报。如果之前 **connect()**未被调用，则 **send()**失败。
- **sendto()**的行为未改变。此函数不限于指定的对等端。
- 函数 **recv()**或 **recvfrom()**仅返回指定对等端已发送的数据。

3.20 关闭数据报套接字

应用可以通过调用 **closesocket()**关闭一个数据报套接字。在函数返回之前，将发生下列操作：

- 立即返回对 **recvfrom()**函数的未完成调用。
- RTCS 丢弃排队等待套接字的数据包，并释放缓冲区。

当 **closesocket()**返回时，套接字句柄无效，应用不能再使用该套接字。

3.21 使用流套接字

3.22 更改流套接字选项

应用可在任何时候更改某些流套接字选项的值。更多详情参见第 7 章"函数参考"中 **setsockopt()** 的列表。

3.23 建立流套接字连接

应用可通过以下方式建立一个流套接字连接:

- 被动方式 — 监听呼入的连接请求（通过调用 **listen()**，紧接着调用 **accept()**）。
- 主动方式 — 生成一个连接请求（通过调用 **connect()**）。

3.23.1 被动建立流套接字连接

通过调用 **listen()**，应用可以被动地将未连接的套接字设定为监听状态，之后本地套接字端点就可以响应单一的传入连接请求。

调用 **listen()**之后，应用调用 **accept()**，该函数将返回一个新的套接字句柄，使应用接受传入连接请求。应用通常在调用 **listen()**后立即调用 **accept()**。应用使用新的套接字句柄进行与指定远程端点的所有通信，直到连接被任一端点关闭。原始套接字保持处于监听状态，并继续被 **socket()**返回的初始套接字句柄引用。

由监听-接收机制创建的新套接字将继承父套接字的配置选项。

3.23.2 主动建立流套接字连接

通过调用 **connect()**，应用可以对函数指定的远程端点主动建立一个流套接字连接。若远程端点未处于监听状态，则 **connect()**返回失败。**connect()**会立即或在超过连接超时套接字选项指定的时间之后返回失败，取决于远程端点的状态。

若远程端点接受此连接, 应用将使用原始套接字句柄实现与该远程端点的所有通信，RTCS 负责维护此连接，直到任一端点将连接关闭。

3.24 获取流套接字名

应用创建了流套接字连接之后，可获取本地端点的标识（通过调用 **getsockname()**）和远程端点的标识（通过调用 **getpeername()**）。

3.25 发送流数据

应用通过调用 **send()**发送流套接字上的数据。函数何时返回取决于发送无等待 (OPT_SEND_NOWAIT)套接字选项的值。应用可通过调用 **setsockopt()**更改这些值。

发送无等待 （无阻塞 **I/O**）	**send()**返回的时机:
FALSE（默认）	TCP 已缓冲所有数据，但不一定已发送。
TRUE	立即返回（返回结果是一个已完全填充或部分填充的缓冲区）。

3.26 接收流数据

应用通过调用 **recv()**接收流套接字上的数据。应用传递给函数一个缓冲区, 供 RTCS 用于存放传入数据。函数何时返回取决于接收-无等待（OPT_RECEIVE_NOWAIT) 和接收-压栈(OPT_RECEIVE_PUSH)）套接字选项的值。应用可通过调用 **setsockopt()**更改这些值。

接收无等待 （无阻塞 **I/O**）	接收压栈 （延迟传输）	**recv()**返回的时机:
FALSE（默认）	TRUE（默认）	满足其一: 接收到数据中的压栈标志。 提供缓冲区中已装满数据。

下一页继续介绍此表...

接收无等待 （无阻塞 **I/O**）	接收压栈 （延迟传输）	**recv()**返回的时机:
		接收超时（默认超时限制为无限长时间）。
FALSE（默认）	FALSE	两者任一: 提供缓冲区中已装满数据。 接收超时。
TRUE	（忽略）	向 TCP 轮询内部接收缓冲区中的任何数据后立即返回。

3.27 缓冲数据

RTCS 单套接字发送缓冲区的大小取决于控制发送缓冲区大小的套接字选项。RTCS 将应用提供的缓冲区中的数据复制到其发送缓冲区。对等端发出数据应答后，RTCS 释放缓冲区内的空间。若缓冲区已满，对 **send()**的调用将被阻塞，直到远程端点应答部分或全部数据后阻塞才能解除。

RTCS 单套接字接收缓冲区的大小取决于控制接收缓冲区大小的套接字选项。当没有正在进行的 **recv()**调用时，RTCS 使用缓冲区保存接收数据。当应用调用 **recv()** 时，RTCS 从其缓冲区中将数据复制到应用缓冲区，以便接收远程端点的其它数据。

3.28 提高流数据的吞吐量

- 仅在需要时在发送数据流的尾部加入压栈标志。
- 指定尽可能大的发送缓冲区和接收缓冲区，以减少应用和 RTCS 的工作量。
- 当调用 **recv()**时，立即再调用一次，以减少 RTCS 必须要复制到其接收缓冲区中的数据量。
- 将发送缓冲区和接收缓冲区的大小指定为最大数据包大小的整数倍。
- 调用 **send()**时的数据量为最大数据包大小的整数倍。

3.29 关闭流套接字

应用可通过调用 **closesocket()**关闭流套接字。SO_LINGER 套接字选项表示套接字的关闭方式：正常关闭或强制关闭（TCP 复位）。默认情况下，函数将立即返回。如果需要使用阻塞方式，可通过 SO_LINGER 套接字选项设置。

未完成的 **send()**和 **recv()**函数调用将立即返回，RTCS 将在 **closesocket()**返回前丢弃套接字接收缓冲区中的任何数据。当 closesocket()返回时，套接字句柄无效，应用不能再使用该套接字。

3.29.1 正常关闭

若套接字将要正常关闭，RTCS 则尝试传送套接字发送缓冲区中的所有数据。默认情况下，RTCS 在远程端点断开后仍保持套接字连接 2 秒钟。如需更长时间，请设置 SO_LINGER 套接字选项。

3.29.2 强制关闭

套接字被强制关闭时，会发生以下动作:

- RTCS 立即丢弃套接字以及套接字内部的发送缓冲区和接收缓冲区。
- 远程端点在发送完缓冲区中的所有数据后立即释放其套接字。

表 3-2. 一览表：套接字函数

函数	说明
accept()	接收下一个呼入的流连接，并复制套接字，以创建一个新的套接字为连接提供服务。
bind()	通过一个端口号来标识本地应用端点。
closesocket()	关闭连接并丢弃套接字。
connect()	为应用端点创建一个流连接，或为数据报套接字设置一个远程端点。
getpeername()	获取一个已连接套接字的对等端地址的和端口号
getsockname()	获取一个已绑定套接字的本地地址和端口号
getsockopt()	获取一个套接字选项的值。
listen()	允许套接字指定的端口接收呼入的流连接。
recv()	在流套接字或数据报套接字上接收数据。
recvfrom()	在数据报套接字上接收数据。
RTCS_geterror()	获取 RTCS 函数返回套接字错误的原因
RTCS_selectall()	等待调用方拥有的任何套接字上发生的活动。
RTCS_selectset()	等待一个套接字集合中任何套接字上发生的活动。
select()	等待指定套接字集合中任何套接字上发生的活动。可区分读/写活动。
send()	在指定了远程端点的流套接字或数据报套接字上发送数据。
sendto()	在数据报套接字上发送数据。
setsockopt()	设定一个套接字选项的值。
shutdownsocket()	不允许继续发送或接收套接字请求。
socket()	创建一个套接字。

3.30 示例

QOTD 服务器设置了一个数据报套接字和一个流套接字。然后该服务器进入无限循环。如果流套接字接收到一个连接请求, 该服务器将接受此请求并发送一条引语。如果数据报套接字接收到数据，该服务器将发送一条引语。

```
sockaddr_in    laddr, raddr;
uint32_t        sock, listensock;
int32_t         length;
uint32_t        index;
uint32_t        error;
uint16_t        rlen;
/* Set up the UDP port (Quote server services port 17): */
laddr.sin_family      = AF_INET;
laddr.sin_port        = 17;
laddr.sin_addr.s_addr = INADDR_ANY;
/* Create a datagram socket: */
sock = socket(PF_INET, SOCK_DGRAM, 0);
if (sock == RTCS_SOCKET_ERROR) {
  printf("\nFailed to create datagram socket.");
  _task_block();
}
/* Bind the datagram socket to the UDP port: */
error = bind(sock, &laddr, sizeof(laddr));
  if (error != RTCS_OK) {
    printf("\nFailed to bind datagram - 0x%lx.", error);
    _task_block();
}
/* Create a stream socket: */
sock = socket(PF_INET, SOCK_STREAM, 0);
if (sock == RTCS_SOCKET_ERROR) {
  printf("\nFailed to create the stream socket.");
  _task_block();
}
/* Bind the stream socket to a TCP port: */
error = bind(sock, &laddr, sizeof(laddr));
if (error != RTCS_OK) {
  printf("\nFailed to bind the stream socket - 0x%lx", error);
  _task_block();
}
/* Set up the stream socket to listen on the TCP port: */
error = listen(sock, 0);
if (error != RTCS_OK) {
  printf("\nlisten() failed - 0x%lx", error);
  _task_block();
}
listensock = sock;
printf("\n\nQuote Server is active on port 17.\n");
index = 0;
for (;;) {
  sock = RTCS_selectall(0);
  if (sock == listensock) {
    /* Connection requested; accept it. */
    rlen = sizeof(raddr);
    sock = accept(listensock, &raddr, &rlen);
    if(sock == RTCS_SOCKET_ERROR) {
    uint32_t rtcserrno = RTCS_get_errno();
    if(rtcserrno == RTCSERR_SOCK_CLOSED) {
        /* Other task must have called closesocket() on the listensock handle.
         * The listensock does not exists, we must not access it's handle anymore.
         */
        handle_closed_listensock();
        _task_block();
```

```
    }
    else {
        printf("\naccept() failed, error 0x%lx",
            RTCS_geterror(listensock));
        continue;
    }
}
    /* Send back a quote: */
    send(sock, Quotes[index], strlen(Quotes[index]) + 1, 0);
    _time_delay(1000);
    shutdown(sock, FLAG_CLOSE_TX);
 } else {
   /* Datagram socket received data. */
   memset(&raddr, 0, sizeof(raddr));
   rlen = sizeof(raddr);
   length = recvfrom(sock, NULL, 0, 0, &raddr, &rlen);
   if (length == RTCS_ERROR) {
     printf("\nError %x receiving from %d.%d.%d.%d,%d",
       RTCS_geterror(sock),
       (raddr.sin_addr.s_addr >> 24) & 0xFF,
       (raddr.sin_addr.s_addr >> 16) & 0xFF,
       (raddr.sin_addr.s_addr >> 8) & 0xFF,
        raddr.sin_addr.s_addr & 0xFF,
        raddr.sin_port);
     continue;
    }
   /* Send back a quote: */
   sendto(sock, Quotes[index], strlen(Quotes[index]) + 1, 0,
     &raddr, rlen);
   }
  ++index;
  if (Quotes[index] == NULL) {
    index = 0;
  }
}
```

第 4 章
点对点驱动程序

4.1 前言

本章描述了如何设置和使用 PPP 点对点驱动程序。

相关信息	参见
本章中提到的数据类型	第 8 章"数据类型"
MQX RTOS	*MQX RTOS* 用户指南 *MQX RTOS* 参考手册
协议	附录 A"协议和政策"
本章中提到的函数的原型。	第 7 章"函数参考"
设置 RTCS	第 2 章 "设置 RTCS"
使用 RTCS 和套接字	第 3 章"使用套接字"

4.2 PPP 及 PPP 驱动程序

PPP 驱动程序符合 RFC 1661 协议，该协议是通过点对点链路传输多协议数据报的标准协议。PPP 驱动程序提供:

- 多协议数据报封装的方法。
- 用于异步串行设备的类 HDLC 帧结构。
- 链路控制协议（LCP），用于建立、配置和测试数据链路的连接。
- 网络控制协议（IPCP），用于建立和配置 IP。

4.2.1 LCP 配置选项

下表列出了驱动程序 PPP 协商的 LCP 配置选项。其中列出了由 RFC 1661 指定的 PPP 驱动程序所使用的默认值。该表还指出了应用可更改哪些选项的默认值。表后还对各选项进行了详细说明。

配置选项		默认值	另请参见
ACCM	异步控制字符映射	0xFFFFFFFF	配置 PPP 驱动程序
ACFC	地址字段和控制字段压缩	FALSE	—
AP	身份验证协议 (用户不能更改 AP 选项自身的默认值，但可更改用来定义身份验证协议的全局变量的默认值。)	(无)	配置 PPP 驱动程序
MRU	最大接收单位	1500	—
PFC	协议字段压缩	FALSE	—

4.2.1.1 ACCM

ACCM 是一个 32 位掩码，其中每个位分别对应从 0x00 到 0x1F 中的一个字符。最低有效位对应于 0x00，最高有效位对应于 0x1F。对于置 1 的位，PPP 驱动程序在每次通过链路发送对应字符时对其进行转义。

由于并非所有处理器都采用相同的位序，因此我们将第 0 位定义为最低有效位。

驱动程序将转义字符作为两个字节发送，顺序如下：

- HDLC 转义字符（0x7D）
- 第 5 位翻转后的转义字符

例如，若 ACCM 第 0 位为 1，通过链路发送的每个 0x00 字节都转义成 0x7D 和 0x20 两个字节发送出去。

PPP 驱动程序始终坚持将此 ACCM 作为链路两端的最小 ACCM。

应用可以更改 ACCM 的默认值。例如，若在链路上使用 XON/XOFF 流控制，应用应将 ACCM 设为 0x000A0000，这样的话，每当 XON (0x11)和 XOFF (0x13)出现在帧内容中时，将对其进行转义。

4.2.1.2 ACFC

ACFC 的默认值为 FALSE。因此 PPP 驱动程序不对 PPP 帧中的"地址"字段和"控制"字段进行压缩。如果 ACFC 变为 TRUE, 驱动程序将省略这些字段并始终假定它们为 0xFF （"地址"字段） 和 0x03 （"地址"字段)。为避免混淆， 当"协议"字段压缩被启用 (即 PFC 配置选项为 TRUE) 且第一个"数据"字段的八位组为 0x03 时，RFC 1661 （PPP） 不允许将 0x00FF 作为"协议"字段的值 (即协议号)。

PPP 驱动程序始终尝试协商 ACFC。

4.2.1.3 AP

在某些连接中，对等端必须进行自我认证后才能交换网络层数据包。PPP 驱动程序支持以下身份认证协议:

- PAP
- CHAP

有关身份认证的更多信息，以及如何更改用来确定认证协议的全局变量的默认值，请参见配置 PPP 驱动程序 。

4.2.1.4 MRU

默认情况下 PPP 驱动程序不进行关于 MRU 的协商, 但随时准备通告任何 1500 字节以内的 MRU。此外， 根据 RFC 791(IP)，PPP 驱动程序从对等端接收的 MRU 不会小于 68 个字节。

4.2.1.5 PFC

PFC 的默认值为 FALSE。因此, PPP 驱动程序不压缩"协议"字段。如果 PFC 的值变为 TRUE，则只要"协议"字段的值 (协议号) 不超过 0x00FF，驱动程序将以单个字节发送"协议"字段。也就是说，如果最高有效字节为 0，则不会被发送。

PPP 驱动程序始终尝试协商 PFC。

4.2.2 配置 PPP 驱动程序

PPP 驱动程序使用一些全局变量，这些变量的默认值根据 RFC 1661 指定。

在为任何链路初始化 PPP 驱动程序之前，应用可通过设定全局变量的值来更改 PPP 驱动程序的配置，也就是在它第一次调用 **PPP_init()** 之前。

更改目标:	默认值:	更改此全局变量:
PPP 驱动程序所需的额外堆栈大小。	0	_PPPTASK_stacksize
CHAP 身份验证信息。	"" NULL NULL	_PPP_CHAP_LNAME _PPP_CHAP_LSECRETS _PPP_CHAP_RSECRETS
PAP 身份验证信息。	NULL NULL	_PPP_PAP_LSECRET _PPP_PAP_RSECRETS
定时器激活时，PPP 驱动程序重启定时器的初始化超时时间（以毫秒为单位）。每当定时器溢出，驱动程序将会使超时时间增大一倍，直到超时时间达到 *_PPP_MAX_XMIT_TIMEOUT*。	3000	_PPP_MIN_XMIT_TIMEOUT
PPP 驱动程序重启定时器的最大超时时间（以毫秒为单位）。	10000	_PPP_MAX_XMIT_TIMEOUT
当 PPP 驱动程序配置一个链路时，LCP 可接受的用于链路双向的最小 ACCM。有关 ACCM 的信息，请参见 ACCM 。	0xFFFF FFFF	_PPP_ACCM
当 PPP 驱动程序协商链路配置时，LCP 在丢弃数据前发送配置请求数据包的次数。	10	_PPP_MAX_CONF_RETRIES
当 PPP 驱动程序关闭一个链路时，在其进入关闭或停止状态前发送终止请求包（未收到对应的终止应答认包）的次数。	2	_PPP_MAX_TERM_RETRIES
当 PPP 驱动程序协商链路配置时，在认为协商无法达成一致前，驱动程序连续发送配置不认可（configure-NAK）包的次数，之后改为发送配置拒绝（configure-reject）包。	5	_PPP_MAX_CONF_NAKS
PPP 驱动程序任务的优先级。 由于用户必须为写入的所有任务指定优先级，因此，RTCS 允许用户更改 PPP 驱动程序任务的优先级，使其适合用户的设计。	6	_PPPTASK_priority

4.2.3 更改身份验证

默认情况下，PPP 驱动程序不使用身份认证协议，但支持以下认证协议:

- PAP
- CHAP

每个协议使用 ID-密码对（PPP_SECRET 结构）。关于此结构的更多详情，请参见第 8 章"数据类型" 中 PPP_SECRET 的列表。

4.2.3.1 PAP

PPP 驱动程序无论是作为客户端还是服务器，都通过两个全局变量控制 PAP:

- _PPP_PAP_LSECRET

  两者之一:

  - NULL（LCP 不允许对等端请求 PAP 协议)。
  - 指针，指向我们向对等端验证自身身份时使用的 ID-密码对 (PPP_SECRET) 。

- _PPP_PAP_RSECRETS

  两者之一:

  - NULL（LCP 不要求对等端进行其身份验证)。
  - 指针，指向一个以 NULL 结尾的数组(PPP_SECRET)，数组元素为验证对等端时使用的所有 ID-密码对 。LCP 要求对等端进行其身份验证。如果对等端拒绝了 PAP 验证协议的协商，LCP 将在链路到达开放状态时立即终止链路。

4.2.3.2 CHAP

PPP 驱动程序通过以下全局变量控制 CHAP:

- _PPP_CHAP_LNAME
- 指针，指向一个以 NULL 结尾的字符串。在服务器端，即服务器名称。在客户端，即客户端名称。
- _PPP_CHAP_LSECRETS

两者之一:

- NULL（LCP 不允许对等端请求 CHAP 协议)。
- 指针，指向一个以 NULL 结尾的数组（PPP_SECRET)，数组元素为我们向对等端验证自身身份时使用的所有 ID-密码对 。
- _PPP_CHAP_RSECRETS

两者之一:

- NULL（LCP 不要求对等端进行其身份验证）。
- 指针，指向一个以 NULL 结尾的数组(PPP_SECRET)，数组元素为验证对等端时使用的所有 ID-密码对 。LCP 要求对等端进行其身份验证。如果对等端拒绝了 CHAP 验证协议的协商，LCP 将在链路到达开放状态时立即终止链路。

4.2.3.3 示例：设置 PAP 和 CHAP 身份验证

4.2.3.4 PAP — 客户端

用户“freescale”的密码为“password1”。

对于客户端的 PAP 身份验证，对如下全局变量进行初始化。

```
char myname[]           = "freescale";
char mysecret[]         = "password1";
PPP_SECRET PAP_secret   = {sizeof(myname)-1,
                           sizeof(mysecret)-1,
                           myname,
                           mysecret};
_PPP_PAP_LSECRET        = &PAP_secret;
```

4.2.3.5 CHAP — 客户端

CHAP 更为灵活，它允许用户在要连接的每个主机上使用不同的密码。用户 arc 有两个帐户，密码如下：

- 在主机 server1 上使用密码 password1。
- 在主机 server2 上使用密码 pawword2。

在客户端初始化全局变量，如下所示：

```
char myname[]                = "freescale";
char server1[]               = "server1";
char mysecret1[]             = "password1";
char server2[]               = "server2";
char mysecret2[]             = "password2";
PPP_SECRET CHAP_secrets[]    = {{sizeof(server1)-1,
                                 sizeof(mysecret1)-1,
                                 server1, mysecret1},
                                {sizeof(server2)-1,
                                 sizeof(mysecret2)-1,
                                 server2,
                                 mysecret2},
                                {0, 0, NULL, NULL}
                               };
_PPP_CHAP_LNAME             = myname;
_PPP_CHAP_LSECRETS          = CHAP_secrets;
```

在此例中，RTCS 运行在主机“server”上，并且有三个用户。

用户	密码
fsl1	*password1*
fsl2	*password2*
fsl3	*password3*

4.2.3.6 PAP — 服务器端

对于服务器端的 PAP 身份验证，对如下全局变量进行初始化。

```
char user1[]               = "fsl1";
char secret1[]             = "password1";
char user2[]               = "fsl2";
char secret2[]             = "password2";
char user3[]               = "fsl3";
char secret3[]             = "password3";
PPP_SECRET secrets[]       = {{sizeof(user1)-1,
                              sizeof(secret1)-1,
                              user1,
                              secret1},
                             {sizeof(user2)-1,
                              sizeof(secret2)-1,
                              user2,
                              secret2},
                             {sizeof(user3)-1,
                              sizeof(secret3)-1,
                              user3,
                              secret3},
                             {0, 0, NULL, NULL}
                            };
_PPP_PAP_RSECRETS          = secrets;
```

4.2.3.7 CHAP — 服务器端

对于服务器端的 CHAP 身份验证，按照如下方法初始化全局变量:

```
char myname[]             = "server";
char user1[]              = "fsl1";
char secret1[]            = "password1";
char user2[]              = "fsl2";
char secret2[]            = "password2";
char user3[]              = "fsl3";
char secret3[]            = "password3";
PPP_SECRET secrets[]      = {{sizeof(user1)-1,
                             sizeof(secret1)-1,
                             user1,
                             secret1},
                            {sizeof(user2)-1,
                             sizeof(secret2)-1,
                             user2,
                             secret2},
                            {sizeof(user3)-1,
                             sizeof(secret3)-1,
                             user3,
                             secret3},
                            {0, 0, NULL, NULL}
                           };
```

示例：设置 PAP 和 CHAP 身份验证

```
_PPP_CHAP_LNAME        = myname;
_PPP_CHAP_RSECRETS     = secrets;
```

4.2.4 初始化 PPP 链路

应用程序在使用 PPP 链路之前，必须先通过调用 **PPP_init()** 来初始化该链路。此函数为链路执行下列任务:

- 分配并初始化内部数据结构和其返回的 PPP 句柄。
- 安装为链路服务的 PPP 回调函数。
- 初始化 LCP。
- 创建为链路服务的发送和接收任务。
- 将链路设为“初始”状态。

4.2.4.1 使用多个 PPP 链路

通过调用每个链路的 PPP_init()， 应用可使用多个 PPP 链路。

4.2.5 获取 PPP 统计数据

若要获取关于 PPP 链路的统计数据，调用 **IPIF_stats()**。

表 4-1. 一览表： 使用 PPP 驱动程序

PPP_init()	针对一个 PPP 链路初始化 PPP 驱动程序（LCP 或 CCP）
PPP_SECRET	身份验证密码
IPIF_stats()	获取关于 PPP 链路的统计数据

4.2.6 示例：使用 PPP 驱动程序

参见第 2 章 “设置 RTCS”

RTCS shell 应用示例中说明了 PPP 服务器和 PPP 客户端的功能。参见%MQX_ROOT%\rtcs\examples\shell 和%MQX_ROOT%\shell\source\rtcs\sh_ppp.c。

第 5 章
RTCS 应用协议

5.1 前言

本章主要描述 RTCS 应用，这些应用为 RTCS 支持的应用层协议实现服务器和客户端的功能。

相关信息	参见
本章中提到的数据类型	第 8 章“数据类型”
MQX RTOS	*MQX RTOS* 用户指南 *MQX RTOS* 参考手册
协议	附录 A“协议和政策”
本章中提到的函数的原型。	第 7 章“函数参考”
设置 RTCS	第 2 章 “设置 RTCS”
使用 RTCS 和套接字	第 3 章“使用套接字”

5.2 DHCP 客户端

根据 RFC 2131 中的描述，动态主机配置协议（DHCP）是一项绑定协议。Freescale MQX RTOS DHCP 客户端基于 RFC 2131。该协议允许 DHCP 客户端在尚未获得 IP 地址和掩码时也能从 DHCP 服务器获取 TCP/IP 配置信息。

默认情况下，当 RTCS DHCP 客户端收到一个服务器发出的提议时，会向提供的 IP 地址发送 ARP 请求来探测网络，以响应其发现者。如果该网络中的主机应答了 ARP 请求，客户端则不接受服务器的提议。反之，它会发送一条消息拒绝服务器的提议，并且送出新的发现。您可以通过确保调用 RTCS_if_bind_DHCP_flagged()时不将 dhcp.h 中定义的标志 DHCP_SEND_PROBE 置位来禁用探测功能。

表 5-1. 一览表：设置 DHCP 客户端

在 RTCS_if_bind_DHCP()使用的选项列表中添加下列各项：	
DHCP_option_addr()	IP 地址
DHCP_option_addrlist()	IP 地址列表
DHCP_option_int8()	8 位值
DHCP_option_int16()	16 位值
DHCP_option_int32()	32 位值
DHCP_option_string()	字符串
DHCP_option_variable()	变长选项
RTCS_if_bind_DHCP()	利用 DHCP 获取 IP 地址并绑定到设备接口。
DHCPCLNT_find_option()	针对特定选项类型搜索 DHCP 消息。

5.2.1 示例：设置和使用 DHCP 客户端

参见第 7 章“函数参考” 中的 RTCS_if_bind_DHCP() 。

5.3 DHCPv6 客户端

IPv6 动态主机配置协议（DHCP）允许 DHCP 服务器向 IPv6 节点传送 IPv6 网络地址等配置参数。它能够实现可重用网络地址的自动分配，提高配置灵活性。更多信息，请参见 RFC3315 （http://tools.ietf.org/html/rfc3315） 。

5.3.1 支持的功能

支持如下网络配置选项：

- 客户端 IPv6 地址。支持多达 IP6_IF_ADDRESSES_MAX 个地址。
- DNS 服务器。
- 域搜索列表（默认域）

链路检查功能：

- 客户端带有集成的链路状态检查机制， 允许当设备再度连接相同网络或不同网络时进行地址确认/再绑定。该功能可通过设置客户端初始化参数中的标志 DHCPCLN6_FLAG_CHECK_LINK 启用。此标志的相关信息请参见获取地址/其他配置 章节。

无状态配置:

- 客户端也可运行于所谓的“无状态”模式。在此模式中，IP 地址取自于无状态地址配置，仅 DNS 服务器地址等其他信息取自于 DNS 服务器。该特性可通过设置客户端初始化参数中的标志 DHCPCLN6_FLAG_STATELESS 启用。此标志的相关信息请参见获取地址/其他配置 章节。

5.3.2 获取地址/其他配置

若要从 DHCPv6 服务器获取 IP 地址，通过 DHCPCLN6_init() 函数启动客户端，该函数带有 DHCPCLN6_PARAM_STRUCT 参数 。客户端启动后，将自动从服务器获取网络配置。以下为 DHCPCLN6_PARAM_STRUCT 的相关成员:

支持下列标志:

flags 成员

DHCPCLN6_FLAG_STATELESS 的相关成员- 若此标志置位，DHCPv6 客户端仅向服务器请求除 IP 地址外的其他附加信息 (DNS 前缀、DNS 服务器 IP 地址等)。DHCPCLN6_FLAG_CHECK_LINK 的相关成员- 若此标志置位，客户端将检查其运行接口的链路状态。如果链路丢失后又重获，将会进行 CONFIRM/REPLY 消息交换。

interface 成员

该变量是 DHCP 客户端启动的 RTCS 接口的句柄。对该变量的设置为强制性。如果向 DHCPCLN6_init()函数传递了一个无效句柄，该操作将失败。

5.3.3 释放获取的地址

若要释放 DHCPv6 客户端获取的任何地址，调用 RTCS6_if_unbind_addr() 函数。如果未通过 DHCP 绑定任何地址，客户端会自动停止。

5.3.4 停止客户端

若要停止 DHCPv6 客户端并释放所有获取的地址，调用函数 DHCPCLN6_release() ，以客户端句柄（DHCPCLN6_init() 的返回值）为参数。

5.4 DHCP 服务器

DHCP 服务器向请求网络地址和初始化参数的客户端主机分配网络地址和传送初始化参数。Freescale RTCS DHCP 服务器基于 RFC2131 。

默认情况下，RTCS DHCP 服务器在向客户端发布地址之前会先对所请求 IP 地址的网络进行探测。如果服务器接收到响应，则其将发送一个 NAK 回复，并等待客户端请求新的地址。将 DHCPSVR_FLAG_DO_PROBE 标志传递至 DHCPSRV_set_config_flag_off()以禁用探测。

表 5-2. 一览表：使用 DHCP 服务器

在 **DHCPSRV_ippool_add()**使用的选项列表中添加下列各项：	
DHCP_option_addr()	IP 地址
DHCP_option_addrlist()	IP 地址列表
DHCP_option_int8()	8 位值
DHCP_option_int16()	16 位值
DHCP_option_int32()	32 位值
DHCP_option_string()	字符串
DHCP_option_variable()	变长选项
DHCPSRV_init()	创建 DHCP 服务器
DHCPSRV_ippool_add()	向 DHCP 服务器分配一个 IP 地址块

5.4.1 示例：设置和修改 DHCP 服务器

参见第 7 章“函数参考” 中的 DHCPSRV_init() 。

5.5 Echo 服务器

Echo 服务器实现符合 Echo 协议（RFC 862）的服务器。Echo 服务将其收到的任何数据都发回至原始的源。

若要启动 Echo 服务器，应用应调用 **ECHOSRV_init()** 。

Echo 服务器与一个客户端通信。RTCS 中含有可用来与 Echo 服务器通信的 Echo 客户端应用。

5.6 Echo 客户端

Echo 客户端可为 RFC 862 Echo 服务器实现一个客户端。Echo 客户端可向服务器发送一个数据，并接收从服务器返回的数据，然后将发出的数据与收到的数据进行比较。若要连接到一个服务器，应用应调用 ECHOCLN_connect()函数。当连接建立后，使用 ECHOCLN_process()来交换数据。

5.7 FTP 客户端

若要发起一个 FTP 会话，应用应调用 **FTP_open()** 。FTP 会话开始后，客户端通过函数 **FTP_command()**和 **FTP_command_data()** 向 FTP 发出命令。客户端通过调用 **FTP_close()** 关闭 FTP 会话。

5.8 FTP 服务器

文件传输协议（FTP）是一种网络协议，允许用户在主机之间通过 TCP 连接传输文件。该协议在命令端口接收命令，并通过主动式或被动式数据连接传输数据。支持“用户名/密码”基本用户身份验证方式，也可用来为每个用户分别指定根目录。

5.8.1 使用 FTP 客户端通信

FTPSRV 支持如下命令:

- ABOR - 中止当前的文件传输。
- APPE [filename] - 向文件[filename]附加数据。
- CWD [path] - 将工作目录改为[path]。
- CDUP - 将工作目录改为向上一级。
- DELE [filename] - 删除文件[filename]。
- EPSV - 扩展被动模式（IPv6)。
- EPRT - 扩展端口命令（IPv6)。
- FEAT - 列举服务器的功能特性。
- HELP - 显示服务器的帮助文档（命令列表)。
- LIST [dirname] - 列举目录[dirname]中的文件。
- MKDIR [dirname] - 创建目录[dirname]。
- MKD - 同 MKDIR。
- NLST [dirname] - 列举目录[dirname]中的文件。
- NOOP - 空操作（空操作命令)。
- PASS [password] - 输入密码。
- PASV - 被动传输模式。
- PORT [host-port] - 设置用于传输数据的主机和端口。
- PWD - 打印工作目录。
- QUIT - 断开与服务器的连接。
- RMDIR [dirname] - 删除目录[dirname]。
- RMD - 同 RMDIR。
- RETR [filename] - 从服务器检索文件[filename]。

- RNFR [filename] - 将文件[filename]重命名。
- RNTO [filename] - 重命名为[filename]。
- SITE - 与站点有关的信息。
- SIZE [filename] - 获取[filename]大小。
- STOR [filename] - 保存文件[filename]到服务器。
- SYST - 获取系统名。
- TYPE [type] - 将传输数据的类型设为[type]。
- USER [username] - 登录用户[username]。
- XCUP - 同 CDUP。
- XCWD - 同 CWD。
- XMKD - 同 MKDIR。
- XPWD - 同 PWD。
- RMD - 同 RMDIR。

5.8.2 编译时配置

FTP 服务器编译时的默认配置可通过少量的宏来设置。关于这些配置的默认值，可在%MQX_PATH%\rtcs\source\include\rtcscfg.h 文件中查询。如需更改任何选项，请向项目的 user_config.h 文件中添加所需的定义指令。

- FTPSRVCFG_DEF_SERVER_PRIO：服务器任务的默认优先级。该值在 FTP 服务器创建主任务和会话任务时使用，通过将服务器初始化结构的 server_prio 成员设为非零值可对其覆盖。该宏默认值为 TCP/IP 任务优先级减 1。
- FTPSRVCFG_DEF_ADDR：服务器的默认 IPv4/IPv6 地址。如果此地址设置了一个不同的值，而并非由服务器初始化结构中的 ipv4_addres 或 ipv6_address 成员（取决于所选的地址系列）设置，服务器会监听此地址。该宏默认值为 INADDR_ANY。
- FTPSRVCFG_DEF_SES_CNT: 默认最大会话数。该值限定了服务器会话或连接的最大数目。每当客户端建立一个新的连接时就会创建一个新的会话。通过设置服务器初始化结构的 max_ses 成员可覆盖此参数的值。默认值为 2 个会话。
- FTPSRVCFG_TX_BUFFER_SIZE：套接字发送缓冲区的大小，以字节数计。该选项在运行时不可覆盖。默认值为 1460 字节。
- FTPSRVCFG_RX_BUFFER_SIZE：套接字接收缓冲区的大小，以字节数计。该选项在运行时不可覆盖。默认值为 1460 字节。
- FTPSRVCFG_TIMEWAIT_TIMEOUT：套接字上进行发送/接收操作的超时值，以毫秒计。该选项在运行时不可覆盖。默认值为 1000ms。
- FTPSRVCFG_SEND_TIMEOUT: 服务器套接字的超时值，以毫秒计，在运行时无法更改。默认值为 500ms。
- FTPSRVCFG_CONNECT_TIMEOUT：FTP 服务器套接字建立连接的硬超时，以毫秒计，在运行时无法更改。默认值为 5000ms。

- FTPSRVCFG_RECEIVE_TIMEOUT：超时值，用于 recv()函数。超过此超时值后 recv()返回其所有的任何数据，该值在运行时无法更改。默认值为 50ms。
- FTPSRVCFG_IS_ANONYMOUS：定义在运行服务器特权命令时是否需要登录/密码的宏。如要设为 0 (默认值)，则需使用密码登录。否则无需进行身份认证。

5.8.3 基本用法

若要成功启动 FTP 服务器，用户仅需遵循两个步骤:

1. 创建并填充类型结构 FTPSRV_PARAM_STRUCT，采用要求的服务器设置。所有参数 (除根目录外) 均为可选参数。可将任何参数都设置为 0，服务器会使用默认值。
2. 启动服务器，使用函数 FTPSRV_init()以及上一步骤中创建的参数。这两个步骤均可通过%MQX_PATH%\shell\source\rtcs\sh_ftpsrv.c 中的示例说明。关于服务器参数结构的描述，请参阅 FTPSRV_PARAM_STRUCT 章节。

5.9 LLMNR 服务器

本地链路多播名称解析（LLMNR）是一种网络协议，允许 IPv4 和 IPv6 主机对同一本地链路上的其他主机进行名称解析。RFC 4795 中对 LLMNR 进行了定义。该协议基于域名系统（DNS）数据包格式，但在独立于 DNS 的端口上操作。Microsoft Windows®操作系统为 LLMNR 提供本机支持。因此，无需安装其他计算机工具。RTCS 支持 LLMNR 服务器。

若要启动 LLMNR 服务器，用户仅需遵循两个步骤:

1. 创建 LLMNRSRV_PARAM_STRUCT 结构并填写要求的服务器设置，如主机名称表和 LLMNR 服务器监听的接口句柄。用户可将任何其他参数都设置为 0，服务器将使用默认值。
2. 通过函数 LLMNRSRV_init()启动服务器，并使用上一步骤中创建的参数结构。这两个步骤均可通过%MQX_PATH%\shell\source\rtcs\sh_llmnrsrv.c 文件中的示例说明。

使用 LLMNRSRV_release()函数可停止服务器。

5.10 HTTP 服务器

超文本传输协议（HTTP） 服务器是一种简单的 web 服务器，用来处理、评估和响应 HTTP 请求。它根据配置和呼入的客户端请求返回静态的文件系统内容，如 Web 页面、样式表、图像或回调例程动态生成的内容。HTTPSRV 应用程序支持 RFC1945 定义的 HTTP 1.0 协议。此外，还能实现以下 HTTP 1.1 功能特性:

- GET、POST 和 HEAD 请求
- CGI 脚本 RFC3875
- 类 ASP 服务器端包含（这些命令中的参数在'<%'和'%>'内）
- 基本身份验证
- HTTP 持久连接（keep-alive）
- 百分号编码 URI
- 缓存控制
- 多根目录（别名）
- 分块传输编码

服务器为每个来自客户端的呼入连接创建一个单独的任务和一个内部数据结构。即所谓的"会话"，下文中将对此详细描述。若会话进程完毕(向客户端发送一个响应)时 keep-alive 选项处于禁用状态，则客户端的连接将被关闭且该会话将被销毁。若 keep-alive 被启用，连接将保持开放，服务器就会等待客户端发送下一个请求。由于无需重建连接，在传输多个小文件时采用这种方式可以加快传输速度。

5.10.1 缓存控制

服务器执行简单的 HTTP 缓存控制指令，也就是通过网络浏览器缓存静态文件，在重新加载网页时则无需更新。可缓存的文件类型（指令 Cache-Control: max-age=3600）如下所列:

- js
- css
- gif
- htm
- jpg
- png
- html

不可缓存受身份验证保护的文件(使用指令 Cache-Control: no-store)。文件在缓存中保存的时间通过宏 HTTPSRVCFG_CACHE_MAXAGE 的值确定。默认为 3600 秒。关于缓冲控制机制的更多详情，请参见 RFC2616，14.9 节 。

5.10.2 支持的 MIME 类型

支持以下 MIME 类型:

- text/plain
- text/html
- text/css
- image/gif
- image/jpeg
- image/png
- image/svg+xml
- application/javascript
- application/xml
- application/zip
- application/pdf
- application/octet-stream

如果没有其他可用的 MIME 类型，默认类型为 application/octet-stream。

5.10.3 别名

别名机制使用户可以访问服务器根目录子文件夹以外的文件系统和文件夹。每个别名目录都具有一个用户定义的名称，客户端可以通过这些名称访问别名目录。本示例说明了在 MQX RTOS 中如何访问挂载为 c:盘的 USB 大容量存储设备中的文件。选择的名称是“usb”，所有文件均可通过此链接获得：http://SERVER_IP_ADDRESS/usb/。

示例代码:

```
HTTPSRV_ALIAS http_aliases[] = {
    {"/usb/", "c:\\"},
    {NULL, NULL}
};

//Initialization code for RTCS
HTTPSRV_PARAM_STRUCT params;
_mem_zero(&params, sizeof(HTTPSRV_PARAM_STRUCT));

params.root_dir = "tfs:";
params.alias_tbl = (HTTPSRV_ALIAS*)http_aliases;

server = HTTPSRV_init(&params);
if(!server)
{
    printf("Error: HTTP server init error.\n");
}
```

5.10.4 编译时配置

HTTP 服务器编译时的默认配置可通过少量的宏来设置。关于这些配置的默认值，可在%MQX_PATH%\rtcs\source\include\rtcscfg.h 文件中查询。如需更改任何选项，请向项目的 user_config.h 文件中添加所需的定义指令。

- HTTPSRVCFG_DEF_ADDR: 服务器的默认 IPv4/IPv6 地址。如果此地址设置了一个不同的值，而并非由服务器初始化结构中的 ipv4_address 或 ipv6_address 成员（取决于所选的地址族）设置，服务器会监听此地址。该宏默认值为 INADDR_ANY。
- HTTPSRVCFG_DEF_SERVER_PRIO: 服务器任务的默认优先级。该值在 HTTP 服务器创建主任务、会话和脚本处理任务时使用。通过将服务器初始化结构的 server_prio 成员设为非零值可对其进行覆盖。该宏默认值为 RTCS TCP/IP 任务优先级减 1。
- HTTPSRVCFG_DEF_PORT 默认监听端口。通过将服务器初始化结构的端口成员设为非零值可对其进行覆盖。该宏默认值为 80。
- HTTPSRVCFG_DEF_INDEX_PAGE: 默认的索引页。该宏指定了在客户端请求根目录时('/')作为响应发送的网页的名称。通过设置服务器初始化结构的 index_page 成员可对其进行覆盖。默认的索引页为"index.htm"。
- HTTPSRVCFG_DEF_SES_CNT: 默认最大会话数。该值限定了服务器创建的会话或连接的最大数目。每当客户端建立一个新的连接时就会创建一个新的会话。通过设置服务器初始化结构的 max_ses 成员可覆盖此参数的值。默认值为 2 个会话。
- HTTPSRVCFG_SES_BUFFER_SIZE: 会话缓冲区的默认大小，以字节数计。该缓冲区用来存放会话所需的所有数据。该选项在运行时不可覆盖。该宏的默认值设为 1360 字节，最小为 512 字节。
- HTTPSRVCFG_DEF_URL_LEN: URL 的默认最大长度，以字符计。通过设置服务器初始化结构的 max_uri 成员可设置此参数的值。当 URL 超出此长度，将向客户端发送代码 414（Request-URI Too Long）的响应。该宏默认值为 128。
- HTTPSRVCFG_MAX_SCRIPT_LN:（CGI 和 SSI）脚本名的最大长度，以字符计。名称超过此长度的任何脚本都将被忽略。该宏默认值为 32。
- HTTPSRVCFG_KEEPALIVE_ENABLED: 定义是否启用/禁用 HTTP keep-alive 的宏。该宏默认值为 0（禁用）。该选项在运行时无法更改。
- HTTPSRVCFG_KEEPALIVE_TO: 使用 keep-alive 时默认的会话超时。该值决定了服务器在上一请求成功处理后对下一请求的等待时间，以毫秒计。该值在运行时不可覆盖。该宏默认值为 200 ms。
- HTTPSRVCFG_SES_TO: 会话超时，以毫秒计。该值决定了会话在被终止前可保持不活动状态的最大时长。该选项在运行时无法更改。默认值为 20000 ms (20 s)。

- HTTPSRVCFG_TX_BUFFER_SIZE：套接字发送缓冲区的大小，以字节数计。该选项在运行时不可覆盖。默认值为 1460 字节。
- HTTPSRVCFG_RX_BUFFER_SIZE：套接字接收缓冲区的大小，以字节数计。该选项在运行时不可覆盖。默认值为 1460 字节。
- HTTPSRVCFG_TIMEWAIT_TIMEOUT: 套接字上进行发送/接收操作的超时值，以毫秒计。该选项在运行时不可覆盖。默认值为 1000ms。
- HTTPSRVCFG_RECEIVE_TIMEOUT: 超时值，用于 recv()函数。超过此超时值后 recv()返回其所有的任何数据，该值在运行时无法更改。默认值为 50ms。
- HTTPSRVCFG_CONNECT_TIMEOUT：HTTP 服务器套接字建立连接的硬超时，以毫秒计。该值在运行时无法更改。默认值为 5000ms。
- HTTPSRVCFG_SEND_TIMEOUT: 服务器套接字的超时值。该选项在运行时无法更改。默认值为 500ms。

5.10.5 基本用法

若要成功启动 HTTP 服务器，用户必须遵循两个步骤:

1. 创建并填写类型结构 HTTPSRV_PARAM_STRUCT 采用要求的服务器设置。所有参数均为可选参数。可将任何参数都设置为 0，服务器会使用默认值。
2. 启动服务器，使用函数 HTTPSRV_init()与上一步骤中创建的参数。

这两个步骤均可通过%MQX_PATH%\rtcs\examples\httpsrv 文件夹中的示例说明。关于服务器参数结构的描述，请参阅 HTTPSRV_PARAM_STRUCT 章节。

5.10.6 使用 CGI 回调

如果想在应用中使用 CGI，则必须为每个“脚本”创建一个函数。然后每当客户端请求名称与函数标签相同的 CGI 文件时，都会调用这个函数。指向这些函数的指针必须存放在类型为 HTTPSRV_CGI_LINK_STRUCT 的数组中，该结构必须作为服务器参数结构的组成部分，通过指针 cgi_lnk_tbl 传递给服务器。

SSI 或 CGI 有两种处理方式:

- 单任务：在同一任务中一个接一个地处理各脚本。
- 多任务：在独立的任务中分别处理每个脚本。

单任务处理 (串行处理)：在服务器启动时创建一个任务以处理所有用户脚本。该任务的栈空间由服务器参数结构中的 script_stack 变量决定。一个会话向该任务发出消息，并在脚本将被执行时运行脚本。该会话保持阻塞，直到脚本运行结束。此方法在脚本的栈大小设为零的情况下使用，此设置位于 HTTPSRV_SSI_LINK_STRUCT 或 HTTPSRV_CGI_LINK_STRUCT。

多任务处理 (并行处理): 与之前的情况类似，在服务器启动时创建一个任务，但这个任务有一个空间最小的栈。若在会话处理过程中遇到了脚本，将会向此任务发送一条消息。系统不运行用户回调函数，而是创建一个新的已分离任务，其栈大小设置为 CGI/SSI 链接结构中的值。用户回调函数则运行于这个新任务中。这样允许脚本处理任务即时读取另一个消息，无需等待。

有了并行处理方式，可以轻松地实现某些更复杂的应用程序，比如通过 CGI 上传大文件等。此方法在脚本的栈大小在脚本表中设为非零值的情况下使用。

您也可以将两者结合使用。栈大小设为零的回调函数在脚本处理任务中处理，通过 script_stack 变量设置栈大小。如果某些回调函数在脚本表中有非零的栈，则在独立任务中处理。

示例:

```
/* First the table of CGI callbacks is created*/
static _mqx_int cgi_ipstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_icmpstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_udpstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_tcpstat(HTTPSRV_CGI_REQ_STRUCT* param);
static _mqx_int cgi_rtc_data(HTTPSRV_CGI_REQ_STRUCT* param);

const HTTPSRV_CGI_LINK_STRUCT cgi_lnk_tbl[] = {
    { "ipstat",         cgi_ipstat,   1500},
    { "icmpstat",       cgi_icmpstat, 1500},
    { "udpstat",        cgi_udpstat,  1500},
    { "tcpstat",        cgi_tcpstat,  1500},
    { "rtcdata",        cgi_rtc_data, 0},
    { 0, 0 }    // DO NOT REMOVE - last item - end of table
};

 /* Then table is saved in parameters structure and server is started */

 HTTPSRV_PARAM_STRUCT params;
 uint32_t server;

 _mem_zero(&params, sizeof(params));

  /* Every time client request i.e., file rtcdata.cgi function cgi_rtc_data is called.*/
  params.cgi_lnk_tbl = (HTTPSRV_CGI_LINK_STRUCT*) cgi_lnk_tbl;
  server = HTTPSRV_init(&params);
```

在用户 CGI 函数中，必须采取如下步骤:

1. 检查请求方法 (GET 或 POST)。
2. 创建一个变量，类型为 HTTPSRV_CGI_RES_STRUCT，后文中称之为 “response”。
3. 从客户端读取数据, 使用 httpsrv_cgi_read()函数。所有数据必须在向客户端发送 response 前读取。
4. 填充 response 结构中的变量。这是您向客户端发送数据的必要步骤。所有成员都是必要成员。
5. 写入数据，使用函数 httpsrv_cgi_write()。
6. 返回 response 的 content_length。

第一次调用函数 httpsrv_cgi_write()后，HTTP 服务器自动生成 HTTP 标头。如果您想要发送更多数据，将 response.data 变量设置为待发送数据的地址，并将数据的字节长度保存在 response.data_length 变量中。每当您调用 httpsrv_cgi_write()时，将会使数据存入会话缓冲区然后发送到客户端。

关于客户端和连接的基本信息，可从传递给每个 CGI 调用函数的类型为 HTTPSRV_CGI_REQ_STRUCT 的参数中读取。关于此结构的详细信息，请参见章节 HTTPSRV_CGI_REQ_STRUCT 。

5.10.7 使用服务器端包含（SSI）回调

服务器端包含一些函数，在每次解析“.shtm”或“ .shtml”文件的过程中遇到特殊序列的字符时调用。这个特殊序列中包括入口标记、函数名 (可带有参数) 和出口标记:

```
<%function_name:parameter%>
```

类似于 CGI，每个用于 SSI 的函数必须先得到声明，并且这些函数的指针及其名称/标签必须保存在 HTTPSRV_SSI_LINK_STRUCT 类型的数组中。该数组作为参数结构中的 ssi_lnk_tbl 变量传递给服务器。

示例:

```
    const HTTPSRV_SSI_LINK_STRUCT fn_lnk_tbl[] = {
     { "usb_status_fn", usb_status_fn },
     { 0, 0 }
    };

    uint32_t server;
    HTTPSRV_PARAM_STRUCT params;

   _mem_zero(&params, sizeof(params));
   params.ssi_lnk_tbl = (HTTPSRV_SSI_LINK_STRUCT*)fn_lnk_tbl;
   server = HTTPSRV_init(&params);
```

当从服务器端进行写操作时, 应包含发送给客户端的响应, 并使用 httpsrv_ssi_write() 函数。

5.10.8 使用 CyaSSL 的安全 HTTP

HTTPSRV 支持 HTTPS 协议。若要启用 HTTPSRV 中的 SSL，您必须传递 HTTPSRV_SSL_STRUCT 结构的有效指针，作为 HTTPSRV_PARAMS_STRUCT 中的参数。关于代码示例，参见%MQX_ROOT%\rtcs\examples\httpsrv 下的 HTTPSRV +SSL 工程。

5.10.9 分块传输编码

MQX RTOS 4.1.2 版之后的版本在 HTTPSRV 中支持分块传输编码。此功能允许在 HTTP 的持久连接模式启用后发送总体大小未知的数据 (无内容长度)。激活此功能仅需调用 HTTPSRV_cgi_write() ，响应参数 content_length 设为-1。后续每次调用 HTTPSRV_cgi_write() 会为客户端生成一个 data_length 大小为的数据块。

若要终止向客户端传输数据及发送信号，调用 HTTPSRV_cgi_write()其中 data_length 设为 0，而 data 设为 NULL。关于分块传输编码的更多详情，请参见 RFC2616，3.6 节.

示例:

```
static _mqx_int cgi_test(HTTPSRV_CGI_REQ_STRUCT* param)
{
   HTTPSRV_CGI_RES_STRUCT response = {0};

   response.ses_handle = param->ses_handle;

   response.data = "Th";
   response.data_length = strlen(response.data);
   response.content_length = -1;
   HTTPSRV_cgi_write(&response);

   response.data = "is is";
   response.data_length = strlen(response.data);
   HTTPSRV_cgi_write(&response);

   response.data = " test\r\n\r\nstring.";
   response.data_length = strlen(response.data);
   HTTPSRV_cgi_write(&response);

   response.data = NULL;
   response.data_length = 0;
   HTTPSRV_cgi_write(&response);
}
```

5.10.10 HTTP 服务器存储器空间要求

HTTP 服务器应用的不同组件具有不同的存储器空间要求。所有标注星号 (*) 的项均为用户可配置项， 因此由您的应用决定。表中列出了各默认值。

表 5-3. 服务器任务

耗用存储器（字节）	
堆栈	1500
上下文	148
总计	1648

HTTPSRV 必备组件中的服务器任务，服务器启动后就会自动创建。没有用户可用的配置能够影响其存储器耗用量。根据在用户提供的 HTTPSRV_PARAM_STRUCT 中启用的地址系列创建一个或两个监听套接字（IPv4 和 IPv6 各一）。

表 5-4. 会话任务

耗用存储器（字节）	
数据缓冲区*	1360
套接字缓冲区*	2920
上下文	140
堆栈**	3000
URL	129
总计	7549

为确保服务器正常运行，至少需要一个会话。一旦出现呼入的 HTTP 请求，即创建会话。数据缓冲区大小通过宏 HTTPSRVCFG_SES_BUFFER_SIZE 定义。套接字缓冲区受到 HTTPSRVCFG_TX_BUFFER_SIZE 和 HTTPSRVCFG_RX_BUFFER_SIZE 宏的影响。URL 的最大长度通过 HTTPSRVCFG_DEF_URL_LEN 定义。在与客户端通信时，每个会话都会创建一个套接字。

提示

用户无法配置会话栈的大小，但可在 httpsrv_prv.h（HTTPSRV_SESSION_STACK_SIZE）中进行更改。您可能需要改变该值，因为默认值都是针对最差情况（使用最差编译器调试配置）而选。

危险

请注意，如果会话栈的空间太小，会话栈很可能会溢出并使您的应用崩溃或出现硬故障！

表 5-5. 脚本处理器任务

耗用存储器（字节）	
堆栈	850
消息队列	192
总计	1042

脚本处理器任务仅在用户提供的 HTTPSRV_PARAM_STRUCT 中出现 CGI/SSI 表指针时创建。其栈大小可通过服务器参数结构中的 script_stack 变量指定。也可通过在 HTTPSRV_CGI_LINK_STRUCT 中指定各堆栈的大小来创建多个脚本处理器。更多详情，请参见关于 CGI/SSI 处理的章节 。也有为 API 调用接收而创建的消息队列 。

表 5-6. WebSocket 任务

耗用内存（字节）	
数据缓冲区*	1360
套接字缓冲区*	2920
上下文	148
堆栈**	3000
总计	7428

WebSocket 是可选的 HTTPSRV 组件。可通过将宏 HTTPSRVCFG_WEBSOCKET_ENABLED 设为 1 启用。数据缓冲区大小通过宏 HTTPSRVCFG_SES_BUFFER_SIZE 定义，而套接字缓冲区大小通过宏 HTTPSRVCFG_TX_BUFFER_SIZE 和 HTTPSRVCFG_RX_BUFFER_SIZE 定义。每个 WebSocket 连接对应一个套接字。

提示

用户无法配置 WebSocket 任务栈的大小，但可在 httpsrv_prv.h（HTTPSRV_SESSION_STACK_SIZE）中对其更改。您可能需要改变该值，因为默认值都是针对最差情况 (使用最差编译器调试配置) 而选的。

危险

请注意，如果 WebSocket 的栈空间太小，会话栈很可能会溢出并使您的应用崩溃或出现硬故障！

5.11 WebSocket 协议

WebSocket 是一个在 TCP 连接上提供全双工通信通道的协议。该协议包括一个起始握手信息和其后的基本消息帧，基于 TCP 分层，并按照 RFC6455 标准化。RTCS 实现是 HTTPSRV 应用的一部分 (一个插件)。WebSocket 在很大程度上简化了双向 Web 通信和连接管理的复杂性。

每个 WebSocket 连接都会创建一个任务以处理所有的接收和发送操作。从 HTTP 会话中重复使用通信套接字和数据缓冲区，这是 WebSocket 握手的要求。已实现全 UTF-8 数据验证，并具有与新版 Web 浏览器的互操作性。

5.11.1 WebSocket API

WebSocket API 包含两个函数和四个回调:

- 函数 WS_send()用于通过 WebSocket 发送数据。
- 函数 WS_close()用于关闭 WebSocket。
- 回调 on_message 在接收到消息时调用。
- 回调 on_error 在发生错误时调用。
- 回调 on_connect 在创建了新的 WebSocket 连接时调用。
- 回调 on_disconnect 在 WebSocket 连接释放时调用。

5.11.2 以 HTTPSRV 插件的形式创建 WebSocket

若要设置此插件，须完成以下几个步骤:

1. 创建一个类型为 WS_PLUGIN_STRUCT 的结构，并填写用户函数。该结构决定了在发生 WebSocket 事件时调用的函数。定义简单 WebSocket echo 的示例:

```
WS_PLUGIN_STRUCT ws_echo_plugin = {
    echo_connect, //callbacks
    echo_message,
    echo_error,
    echo_disconnect,
    NULL //callback parameter
};
```

2. 创建一个类型为 HTTPSRV_PLUGIN_STRUCT 的结构，以定义 web 服务器插件和其类型。将 echo 回调函数定义为 WebSocket 插件的示例:

```
HTTPSRV_PLUGIN_STRUCT echo_plugin = {
    HTTPSRV_WS_PLUGIN, //plugin type
    (void *) &ws_echo_plugin  //pointer to callbacks
};
```

3. 创建一个类型为 HTTPSRV_PLUGIN_LINK_STRUCT 的结构，使服务器资源链接到服务器插件。将 ws://SERVER_ADDRESS/echo 链接到 echo 插件的示例:

```
HTTPSRV_PLUGIN_LINK_STRUCT plugins[] = {
    {"/echo", &echo_plugin},
    {NULL, NULL}
};
```

4. 将插件数组设置为 HTTP 服务器的初始化参数。将插件结构设置为服务器插件的示例:

```
HTTPSRV_PARAM_STRUCT params;
...
params.plugins = &plugins;
...
HTTPSRV_init(&params);
...
```

每当客户端（即 web 浏览器）请求 URI ws://SERVER_ADDRESS/echo 时，就会创建一个 WebSocket 连接，并且调用回调函数以处理 WebSocket 事件。

5.11.3 通过 WebSocket 发送数据

使用函数 WS_send()来发送数据。该函数具有一个 WS_USER_CONTEXT_STRUCT 类型的参数结构。此结构中的重要变量大多为 handle、data 和 fin_flag。

1. handle 变量 - 该变量是用来标识连接的数字。此变量最初可能是从 on_connected 回调函数的参数中检索到的。
2. data 变量 - 该变量属于 WS_DATA_STRUCT 类型。此变量用来传递要发送到客户端的数据。它包含三个项目：
   - data_ptr - 指向数据的指针。
   - length - 数据的长度。
   - type - 数据类型（文本或二进制）。

fin_flag 变量 - 该变量用来指示消息的结尾。若该值大于零，缓冲区的内容将送入客户端。

警告

> 数据发送时有必要将 fin_flag 变量设为非零值，因为也许这是发往客户端的最后一个数据区块。否则，服务器无法确定用户数据的结尾。您可以在每次调用以下函数时设置 fin_flag：WS_send()，但应注意这种方式可能会影响 WebSocket 的性能。

关于 fin_flag 用法区别的示例（分两个数据区块发送消息“Hello World!”）：

1. 仅为最后一个数据区块设置 fin_flag：
   - 服务器上发送的数据：
     1. 写入数据“Hello”；fin_flag = 0。
     2. 写入数据“World!”；fin_flag = 1。
   - 客户端接收到的消息：
     1. “Hello World!”
2. 为每个数据区块设置 fin_flag：
   - 服务器上发送的数据：
     1. 写入数据“Hello”；fin_flag = 1。
     2. 写入数据“World!”；fin_flag = 1。
   - 客户端接收到的消息：
     1. “Hello”
     2. “World!”

3. 不为任何数据区块设置 fin_flag:
    - 服务器上发送的数据:
        1. 写入数据“Hello”; fin_flag = 0。
        2. 写入数据“World!”; fin_flag = 0。
    - 客户端接收到的消息:
        1. 无 - 数据仍在服务器的缓冲区中，因为 fin_flag 未设置。

5.11.4 从 WebSocket 接收数据

接收数据后，通过回调函数 on_message 与类型为 WS_USER_CONTEXT_STRUCT 的参数处理数据。关于数据的信息（如接收的字节数、数据类型和指针）存储在类型为 WS_DATA_STRUCT 的数据子结构中。

5.11.5 WebSocket 错误处理

如果在与客户端通信的过程中出现错误，将会调用 on_error 回调函数，使用的参数类型为 WS_USER_CONTEXT_STRUCT 。在此结构中，error 变量代表一种错误类型。下面是关于错误代码的说明:

- WS_ERR_OK: 未出现错误。
- WS_ERR_SOCKET: 客户端在没有进行正确关闭握手的情况下终止了连接。
- WS_ERR_BAD_FRAME: 接收到的帧已损坏（错误的关闭原因，保留域有无效值，等等）。
- WS_ERR_BAD_OPCODE: 帧有效，但设置了错误（未知）的帧操作码。
- WS_ERR_BAD_SEQ: 接收到错误的帧序列（延续帧之间夹有数据帧；延续帧之前没有数据帧，等等）。
- WS_ERR_NO_UTF8: 接收的数据类型设为文本数据，但数据不是有效的 UTF-8 序列。
- WS_ERR_SERVER: 服务器错误；服务器内存不足且服务器无法创建任务，等等。

所有错误均为严重错误，服务器在 on_error 回调处理完成后自动关闭连接。

5.11.6 关闭 WebSocket 连接

若要关闭 WebSocket，调用 WS_close()函数。无需在出错情况下关闭 WebSocket，这种情况由服务器自动处理。

5.12 IPCFG — 高级网络接口管理

IPCFG 是一个高级函数的集合，其中打包了一些 RTCS 网络接口处理函数，参见为设备接口绑定 IP 地址 中的描述。IPCFG 系统可用于监控以太网连接状态并自动调用适合的“bind”函数。

在当前的版本中，IPCFG 支持自动绑定静态 IP 地址或者自动更新 DHCP 分配的地址。它可以独立操作，独立运行一个任务或运行于轮询模式。

IPCFG API 函数全部都带有 ipcfg_前缀。有关更多详情，请参见函数参考章节。

IPCFG 的使用过程如下:

1. 按照前述部分创建 RTCS(**RTCS_create()**)。
2. 通过 **ipcfg_init_device()** 初始化网络设备。
3. 使用一个 ipcfg_bind_*xxx* 函数为接口绑定 IP 地址、掩码及网关。IPv6 地址通过 IPv6 无状态自动配置服务自动分配。若要手动添加 IPv6 地址，请使用 **ipcfg6_bind_addr()** 。相关示例请参见 shell/source/rtcs/sh_ipconfig.c: Shell_ipconfig_staticip()。
4. 通过启动链路状态监视任务(**ipcfg_task_create()**)可在以太网线缆重连接时自动重绑定地址。处理该监视任务的另一个方法是在已有任务中周期性地调用 **ipcfg_task_poll()** 。
5. 可使用各种 iocfg_get_xxx 函数获取绑定信息。

通过 shell 中的 ipconfig 命令可以说明 IPCFG 的整体功能。关于其实现，请参见 shell/source/rtcs/sh_ipconfig.c 源程序文件。

IPCFG 的部分功能取决于在 user_config.h 配置文件中启用或禁用了哪些 RTCS 功能特性。该配置一旦被更改，则必须重建 RTCS 库和所有的应用程序。

IPCFG 功能受以下定义影响:

- RTCSCFG_ENABLE_GATEWAYS：必须设置为非零值，以使 IPCFG 能够到达网络中网关之后的设备。若不使用该功能，IPCFG 将会忽略所有与网关相关的数据。
- RTCSCFG_IPCFG_ENABLE_DNS：必须设置为非零值，以使能 IPCFG 中的 DNS 名称解析功能。

- RTCSCFG_IPCFG_ENABLE_DHCP：必须设置为非零值，以使能 IPCFG 中的 DNS 绑定功能。注意，DHCP 也取决于 *RTCSCFG_ENABLE_UDP*。
- RTCSCFG_IPCFG_ENABLE_BOOT：必须设置为非零值，以使能 TFTP 名称处理和 BOOT 绑定功能。

5.13 IWCFG — 高级无线网络接口管理

IWCFG 是一个高级函数集合, 其中打包了某些无线配置管理函数。通常用于设置无线操作专用的网络接口参数，如 ESSID。也可使用 Iwconfig 来显示这些参数。

所有的参数都与设备相关。每个驱动程序都会根据硬件支持情况提供一些参数，这些参数值的范围可能不同。有关详细信息，请参见文档中每个设备的相关页面。

所有 IWCFG API 函数都带有 iwcfg_前缀。有关更多详情，请参见函数参考章节。

IWCFG 的使用过程如下：

1. 按照前述部分创建 RTCS(**RTCS_create()**)。
2. 通过 **ipcfg_init_device()** 初始化网络设备。
3. 初始化 wifi 设备可使用下列命令：

   iwcfg_set_essid()

   iwcfg_set_passphrase()

   iwcfg_set_wep_key()

   iwcfg_set_sec_type()

   iwcfg_set_mode()
4. 使用一个 ipcfg_bind_xxx 函数为接口绑定 IP 地址、掩码及网关。

5.14 SMTP 客户端

简单邮件传输协议是一种互联网标准，设计用于 IP 网络中的电子邮件传输。RTCS SMTP 客户端基于 RFC 5321。MQX RTOS 的实现支持 IPv4 和 IPv6 协议。

5.14.1 发送邮件

发送电子邮件只需要调用 SMTP_send_email 函数。必须设置一个数据类型为 SMTP_PARAM_STRUCT 结构 的结构，并在函数调用前作为第一个参数传递给函数。如果需要详细的错误/投递消息，用户必须为此消息创建一个缓冲区，请将其和其大小分别作为第二个和第三个参数传递给函数。有关 SMTP 客户端功能的更多参考信息，请参见以下函数和数据类型的参考资料:

- SMTP_send_email()
- SMTP_EMAIL_ENVELOPE
- SMTP_PARAM_STRUCT

5.14.2 示例应用

通过一个示例可以演示 RTCS 中 SMTP 客户端的功能。此段代码可在 %MQX_PATH%\shell\source\rtcs\sh_smtp.c 文件中找到。该文件中含有可实现电子邮件 shell 命令的代码，shell 命令可用来发送经由 RTCS shell 身份验证的电子邮件。

5.15 SNMP 代理

简单网络管理协议 (SNMP) 用于管理基于 TCP/IP 的互联网对象。具有 SNMP 代理的主机、网关和终端服务器等对象可根据网络管理站的请求执行网络管理功能。

Freescale MQX RTOS SNMPv1 代理符合下列 RFC 协议:

- RFC 1155
- RFC 1157
- RFC 1212
- RFC 1213

Freescale MQX RTOS SNMPv2c 代理基于下列 RFC 协议:

- RFC 1905
- RFC 1906

5.15.1 编译时配置

SNMP 代理编译时的默认配置可通过少量的宏来设置。关于这些配置的默认值，可在%MQX_PATH%\rtcs\source\include\rtcscfg.h 文件中查询。如需更改任何选项，请向项目的 mqx_sdk_config.h 文件中添加所需的定义指令。

- RTCSCFG_SNMPSRV_SERVER_PRIO：代理任务的默认优先级。该值用于 SNMP 代理创建其后台任务。通过将服务器初始化结构的 server_prio 成员设为非零值可对其覆盖。该宏默认值为 RTCS TCP/IP 任务优先级减 1。
- RTCSCFG_SNMPSRV_BUFFER_SIZE：传入和传出数据包使用的缓冲区大小，以字节为单位。必须至少为 494 字节。默认值为 1。

5.15.2 定义管理信息库（MIB）

在用户应用中, MIB 数据库对象或节点被定义为 RTCSMIB_NODE 结构的树。这些结构中含有指向父节点、子节点和兄弟节点的指针，因此可以有效地实现存储器中的 MIB 树数据库。每个节点结构还指向一个“value”结构 RTCSMIB_VALUE，其中含有实际的 MIB 节点数据或函数指针（对于在运行时生成的值）。MIB 树通常不需要在运行时更改，因此可将节点结构声明为常量并放入只读存储器中。MIB 树中应含有每个对象的所有父节点，因为需要使用节点 ID 来创建 SNMP 消息中发送的 OID。

变量对象

在抄录代码段，用户应为所有可读的叶变量对象提供 RTCSMIB_VALUE 结构的实现方式。该结构的定义如下:

```
typedef struct rtcsmib_value
{
    uint32_t TYPE;
    void *PARAM;
} RTCSMIB_VALUE;
```

在此结构中，用户可指定在应用中检索对象值所使用的类型和方法。每个 MIB 对象实际上附有两种信息类型:

- 一种直接基于 MIB 标准类型，并附于 RTCSMIB_NODE 结构。
- TYPE 信息附于 RTCSMIB_VALUE 结构。该类型值与 PARAM 成员结合使用。更多详情请参见下表。

可写对象

对于每个可写的变量对象，提供 MIB_set_objectname()函数，其中 objectname 为变量对象的名称。

```
uint32_t MIB_set_objectname
(
    void *instance, /* IN */
```

```
    unsigned char *value_ptr, /* OUT */
    uint32_t value_len /* OUT */
)
```

- instance — NULL，如果 objectname 不在表格中或是由 MIB_find_objectname()返回的一个指针。
- value_ptr — 指向 object 设定值的指针。
- value_len — 值的字节长度。

函数 MIB_set_objectname()应返回下述代码之一：

- SNMP_ERROR_noError — 操作成功。
- SNMP_ERROR_wrongValue — 值非法，所以无法赋值。
- SNMP_ERROR_inconsistentValue — 值合法，但因其他原因而无法赋值。
- SNMP_ERROR_wrongLength — value_len 不适用于此对象类型。
- SNMP_ERROR_resourceUnavailable — 资源不够。
- SNMP_ERROR_genErr — 任何其他原因。

示例：

我们想要在应用中使用 ipForwarding (OID 1.3.6.1.2.1.4.1)。此时 OID 等于：iso(1).org(3).dod(6).internet(1).mgmt(2).mib-2(1).ip(4).ipForwarding(1)。下列节点在 MIB 树中出现时，应具有正确的指针指向父节点和子节点：

```
RTCSMIB_NODE MIBNODE_iso_org
RTCSMIB_NODE MIBNODE_dod
RTCSMIB_NODE MIBNODE_internet
RTCSMIB_NODE MIBNODE_mgmt
RTCSMIB_NODE MIBNODE_mib2
RTCSMIB_NODE MIBNODE_ip
RTCSMIB_NODE MIBNODE_ipForwarding
```

叶节点（MIBNODE_ipForwarding）将如下所示：

```
const RTCSMIB_NODE MIBNODE_ipForwarding = {
    1,
    NULL,
    NULL,
    (RTCSMIB_NODE *)&MIBNODE_ip,
    RTCSMIB_ACCESS_READ,
    NULL,
    MIB_instance_zero,
    ASN1_TYPE_OCTET,
    (RTCSMIB_VALUE *)&MIBVALUE_ipForwarding,
    MIB_set_ipforwarding
};
```

MIBVALUE_ipForwarding 的内容为：

```
const RTCSMIB_VALUE MIBVALUE_ipForwarding = {
    RTCSMIB_NODETYPE_INT_FN,
    (void *)MIB_get_ipforwarding
};
```

用于设置值的函数（有效的 ipForwarding 值见 RFC1213 中的说明）：

```
uint32_t MIB_set_ipforwarding(void *dummy, unsigned char *varptr, uint32_t varlen)
{
```

```
    int32_t varval = MIB_int_read(varptr, varlen);

    if (varval == 1)
    {
        _IP_forward = TRUE;
    }
    else if (varval == 2)
    {
        _IP_forward = FALSE;
    }
    else
    {
        return(SNMP_ERROR_wrongValue);
    }
    return(SNMP_ERROR_noError);
}

int32_t MIB_int_read(uint8_t *varptr, uint32_t varlen)
{
    int32_t varval = 0;

    if (varlen)
    {
        varval = *varptr++;
        varlen--;
    }
    while (varlen--)
    {
        varval <<= 8;
        varval += *varptr++;
    }
    return(varval);
}
```

最后，用于读取 RTCS 中 IP 转发值的函数：

```
int32_t MIB_get_ipforwarding(void *dummy)
{
    if (_IP_forward)
    {
        return(1); /* return 1 - Forwarding enabled */
    }
    else
    {
        return(2); /* return 2 - Forwarding disabled */
    }
}
```

关于整个 MIB 树的完整示例代码，可参考%MQX_PATH%\nshell\source\rtcs 文件夹中的 sh_snmpsrv.c 文件。

5.15.3 向客户端应用发送一个陷阱消息

若要向客户端应用发送一个陷阱消息，使用函数 SNMPSRV_send_trap() 。支持 SNMPv2 陷阱。该函数具有以下参数：

- `uint32_t handle` - 句柄，指向函数 SNMPSRV_init() 创建的 SNMP 代理。必要参数。
- `const struct sockaddr *target` - 套接字地址结构，用来描述陷阱消息的目标（IP 地址、地址系列、端口等）。强制参数。

- `SNMPSRV_TRAP_TYPE type` - 陷阱类型，参见 SNMPSRV_TRAP_TYPE description 的有效值。强制参数
- `void *param` - 陷阱发送函数的参数。该参数必须具有下列值之一，否则陷阱消息发送将会失败:
  - 陷阱类型为 SNMPSRV_TRAP_TYPE_USER- 指针，指向 MIB 树中有效的 RTCSMIB_NODE。该节点的值将被发送至客户端，其 OID 值根据其在 MIB 树中的位置定义。
  - 陷阱类型为 SNMPSRV_TRAP_TYPE_LINKDOWN 或 SNMPSRV_TRAP_TYPE_LINKUP- 指针，指向通信设备描述符。可以是任意值, 但系统中的所有设备都必须具有唯一值。该参数值将在陷阱消息中作为 ifIndex (OID 1.3.6.1.2.1.2.2.1.1)发送。
  - 如果陷阱类型为 SNMPSRV_TRAP_TYPE_COLDSTART, SNMPSRV_TRAP_TYPE_WARMSTART 或 SNMPSRV_TRAP_TYPE_AUTHFAIL 该参数将被忽略。

RTCS shell 示例中通过 snmptrap shell 命令演示了陷阱发送。

5.15.4 基本用法

按照如下步骤启动 SNMP 代理:

1. 创建 MIB 树，并将其指针存放在 SNMPSRV_PARAM_STRUCT 中。更多详情参见“定义管理信息库” 章节。
2. 创建并填充 params 结构的剩余部分, 采用要求的服务器设置。除 MIB 树的指针外，其他所有参数均为可选参数。您可将任何参数都设置为 0/NULL，服务器将使用默认值。
3. 通过函数 SNMPSRV_init() 启动服务器，并使用上一步骤中创建的参数。

这些步骤均可通过 %MQX_PATH%\nshell\source\rtcs\sh_snmpsrv.c 文件中的示例说明。

5.16 简单网络时间协议（SNTP）客户端

RTCS 提供一个基于 RFC 2030 的 SNTP 客户端 （简单网络时间协议）。SNTP 客户端提供两个不同的接口。一个接口作为函数调用将时间设置为当前时间，另一个接口启动 SNTP 客户端任务，定期更新本地时间。

表 5-7. 一览表：SNTP 客户端服务

SNTP_init()	启动 SNTP 客户端任务。
SNTP_oneshot()	通过 SNTP 协议设置时间。

5.17 Telnet 客户端

Telnet 客户端可实现一个符合 Telnet 协议规范（RFC 854）的客户端。Telnet 连接可在使用不同字符集的两台计算机之间建立一个网络虚拟终端。服务器主机向发起通信的用户主机提供一项服务。IPv4 和 IPv6 均受支持。

5.17.1 连接和断开 Telnet 服务器

若要连接远程主机和 Telnet 客户端，使用 TELNETCLN_connect()函数。该函数有一个参数 params TELNETCLN_PARAM_STRUCT。TELNETCLN_PARAM_STRUCT 的关键参数为：

- sa_remote_host - sockaddr 结构用来描述远程主机。该结构的内容必须通过 getaddrinfo()函数进行初始化。其中含有远程 IP 地址、端口和地址族等信息。
- fd_in - 一个指针（描述符），其指向的文件为 Telnet 客户端的输出。Telnet 客户端向其读取字符并将发送到服务器，直到 EOF 出现。
- fd_out - 一个指针（描述符），其指向的文件为 Telnet 客户端的输入。Telnet 客户端向远程主机读取字符，并将这些字符写入该文件，知道连接关闭。
- 回调- 一个 TELNETCLN_CALLBACKS_STRUCT 结构，所含指针指向为不同事件调用的函数。更多详情请参见关于此结构类型的描述。

若连接成功，TELNETCLN_connect()函数将返回一个非零数（句柄）。该句柄可作为其他 API 程序的参数使用。

若要检查客户端是否连接成功，使用 TELNETCLN_get_status()函数，其参数为 TELNETCLN_connect()函数创建的句柄。其返回值可能等于 TELNETCLN_STATUS_STOPPED 或 TELNETCLN_STATUS_RUNNING，取决于客户端的状态。

若要使客户端和服务器断开，调用 TELNETCLN_disconnect()函数。该函数有一个参数，为客户端句柄。如果断开成功，返回的值将等于 RTCS_OK。否则，将被设为 RTCS_ERROR。

5.18 Telnet 服务器

Telnet 是互联网或局域网使用的网络协议，通过使用虚拟终端连接提供面向文本的双向交互式通信设施。在基于传输控制协议（TCP）的 8 位字节数据连接中，用户数据在带内与 Telnet 控制信息相互穿插。RFC 854 中定义了 Telnet 协议的规范（https://tools.ietf.org/html/rfc854） 。

5.18.1 编译时配置

Telnet 服务器编译时的默认配置可通过少量的宏来设置。关于这些配置的默认值，可在%MQX_PATH%\rtcs\source\include\rtcscfg.h 文件中查询。如需更改任何选项，请向项目的 user_config.h 文件中添加所需的定义指令。

- RTCSCFG_TELNETSRV_SERVER_PRIO — 服务器任务的默认优先级。该值在 Telnet 服务器创建主任务和会话任务时使用。通过将服务器初始化结构的 server_prio 成员设为非零值可对其覆盖。该宏的默认值为 TCP/IP 任务优先级减 1。
- RTCSCFG_TELNETSRV_SES_CNT — 默认最大会话数。该值限定了服务器会话（或连接）的最大数目。每当客户端建立一个新的连接时，就会创建一个新的会话。通过设置服务器初始化结构的 max_ses 成员可覆盖此参数的值。默认值为 2 个会话。
- RTCSCFG_TELNETSRV_TX_BUFFER_SIZE 套接字发送缓冲区的大小, 以字节数计。该选项在运行时不可覆盖。默认值为 1460 字节。
- RTCSCFG_TELNETSRV_RX_BUFFER_SIZE 套接字接收缓冲区的大小, 以字节数计。该选项在运行时不可覆盖。默认值为 1460 字节。
- RTCSCFG_TELNETSRV_TIMEWAIT_TIMEOUT — 指定服务器套接字保持 TIME-WAIT 状态的时长。默认值为 1000ms。
- RTCSCFG_TELNETSRV_SEND_TIMEOUT — 服务器套接字的超时值。默认值为 5000ms。
- RTCSCFG_TELNETSRV_CONNECT_TIMEOUT — telnet 服务器套接字建立连接的硬超时，以毫秒计。该值在运行时无法更改。默认值为 1000ms。
- RTCSCFG_TELNETSRV_USE_WELCOME_STRINGS — 该宏用来定义在客户端连接/断开时是否向其发送欢迎消息和离开消息 。

5.18.2 基本用法

若要成功启动 Telnet 服务器，用户仅需遵循两个步骤:

1. 创建并填充类型结构 TELNETSRV_PARAM_STRUCT，采用要求的服务器设置。除 shell 和 shell 命令外的所有参数均为可选。您可将任何参数都设置为 0/NULL，服务器将使用默认值。

2. 通过函数 TELNETSRV_init()启动服务器，并使用上一步骤中创建的参数。这两个步骤均可通过%MQX_PATH%\shell\source\rtcs\sh_telnetsrv.c 文件中的示例说明。关于服务器参数结构的描述，请参阅 TELNETSRV_PARAM_STRUCT 。

5.19 TFTP 客户端

TFTP 客户端的 RTCS 实现支持 IPv4 和 IPv6 远程主机，允许两种操作：

1. 从服务器下载一个文件到本地文件系统。
2. 上传一个本地文件到 TFTP 服务器。

编译时配置

TFTP 客户端应用没有编译时配置。

基本用法

首先, TFTP 客户端必须初始化。通过调用 TFTPCLN_connect()函数便可实现。该函数的参数为 TFTPCLN_PARAM_STRUCT，其中含有以下变量：

- sockaddr sa_remote_host - 此参数用来描述 TFTP 服务器。 这是唯一必需的参数。用户可使用 getaddrinfo 函数获取此结构。
- TELNETCLN_DATA_CALLBACK recv_callback - 该函数在从服务器接收数据时调用。它有一个参数，即所接收数据的字节数。
- TELNETCLN_DATA_CALLBACK send_callback - 该函数在向服务器发送数据时调用。它有一个参数，即所发送数据的字节数。
- TELNETCLN_ERROR_CALLBACK error_callback - 该函数在传输数据过程中发生错误时调用。它有两个参数：错误代码和描述错误的字符串。

如果初始化成功, 将会创建一个任务以处理 GET/PUT 请求和 TFTP 超市, 返回值为非零。

若要下载一个文件，调用 TFTPCLN_get()函数。该函数下载一个文件后返回。如果用户在 TFTP 初始化过程中提供了一个接收回调函数，则每收到一个数据包时都会调用此回调函数。此函数有三个参数：一个 TFTPCLN_connect()函数创建的句柄，一个本地文件名和一个远程文件名。有关 TFTPCLN_get()函数的详细信息，请参见第 7 章中的函数描述。

若要向 TFTP 服务器上传一个文件，使用 TFTPCLN_put()函数。该函数上传一个文件后返回。如果用户在 TFTP 初始化过程中提供了一个发送回调函数，则每发送一个数据包时都会调用此回调函数。此函数有三个参数，一个 TFTPCLN_connect()函数创建的句柄，一个本地文件名和一个远程文件名。有关 TFTPCLN_put()函数的详细信息，请参见第 7 章。

如果用户想要从内存（或向内存）发送接收到的数据而非文件，则应使用 io_mem、nmem、pipe 或 npipe。

5.20 TFTP 服务器

TFTP 是用于传输文件的简单协议。其实现位于互联网用户数据报协议（UDP 或数据报）的顶层。其设计宗旨是小巧且易于实现。因此，它缺少常规 FTP 的大部分功能。它能执行的就是从/向远程服务器读取/写入文件。

5.20.1 编译时配置

TFTP 服务器编译时的默认配置可通过少量的宏来设置。关于这些配置的默认值，可在%MQX_PATH%\rtcs\source\include\rtcscfg.h 文件中查询。如需更改任何选项，请向项目的 user_config.h 文件中添加所需的定义指令。

- RTCSCFG_TFTPSRV_SERVER_PRIO — 服务器任务的默认优先级。该值在 TFTP 服务器创建主任务和会话任务时使用。通过将服务器初始化结构的 server_prio 成员设为非零值可对其覆盖。该宏的默认值为 TCP/IP 任务优先级加 1。
- RTCSCFG_TFTPSRV_SES_CNT — 默认最大会话数。该值限定了服务器会话（或连接）的最大数目。每当客户端建立一个新的连接时，就会创建一个新的会话。通过设置服务器初始化结构的 max_ses 成员可覆盖此参数的值。默认值为 2 个会话。

5.20.2 基本用法

若要成功启动 TFTP 服务器，用户仅需遵循两个步骤：

1. 创建并填充类型结构 TFTPSRV_PARAM_STRUCT，采用要求的服务器设置。所有参数（除根目录外）均为可选参数。您可将任何参数都设置为 0/NULL，服务器将使用默认值。
2. 通过函数 TFTPSRV_init()启动服务器，并使用上一步骤中创建的参数。这两个步骤均可通过%MQX_PATH%\shell\source\rtcs\sh_tftpsrv.c 文件中的示例说明。关于服务器参数结构的描述，请参阅 TFTPSRV_PARAM_STRUCT 。

5.21 典型的 RTCS IP 数据包路径

图 5-1 是针对 RTCS 应用中 IP 数据包处理的典型代码路径示意图。此示意图仅演示了常规用途，比如寻找设置断点的适当位置。所列函数均为 RTCS 的内部函数。驱动程序的输入和输出接口仅针对媒体接口驱动软件，如以太网驱动程序。

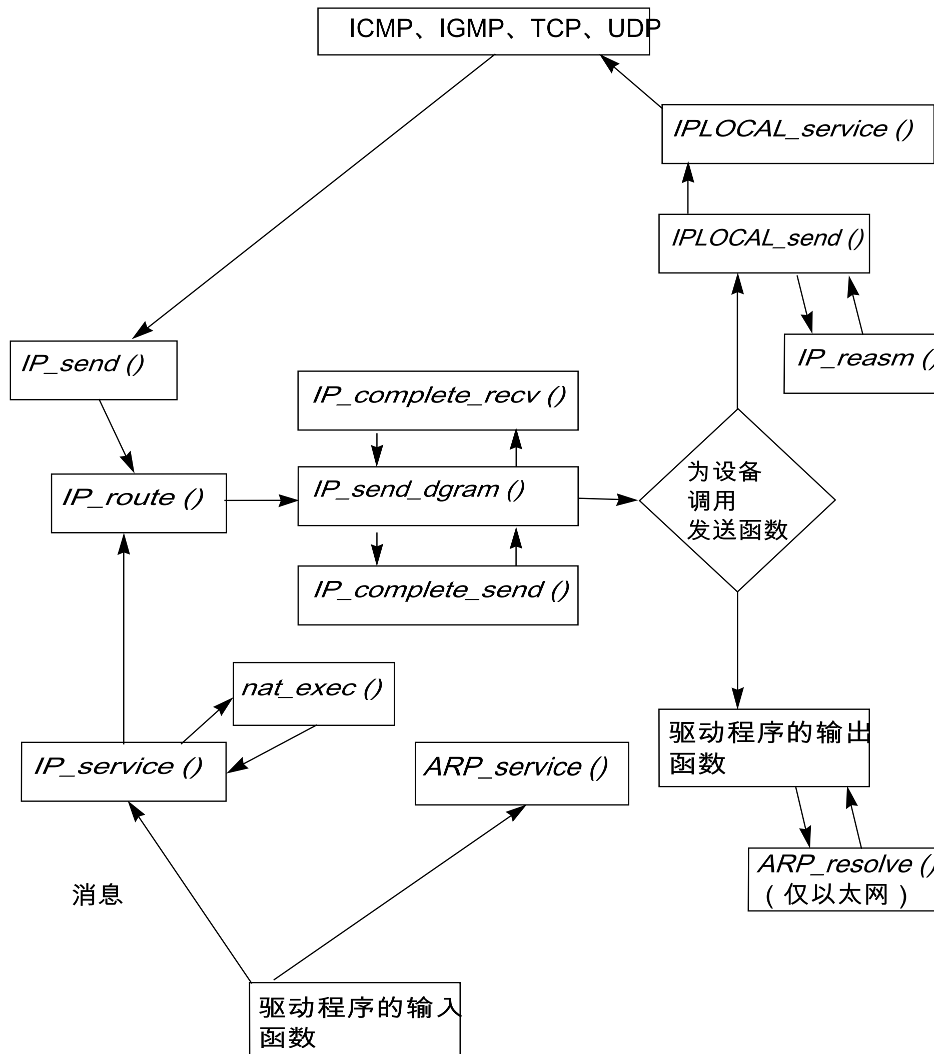

图 5-1. 典型的 RTCS IP 数据包处理路径

接收 **IPoE** 数据包

6字节	6字节	2字节	46 - 1500字节	4字节
目标 地址	源 地址	类型	数据	FCS

图 5-2. 用于接收数据的典型 RTCS IPoE 代码路径

发送 **IPoE** 数据包

图 5-3. 用于发送数据的典型 RTCS IPoE 代码路径

第 6 章
重新编译

6.1 重新编译 RTCS 的原因

如果进行了以下操作，就需要重新编译 RTCS:

- 改变了编译器选项，如优化等级。
- 改变了 RTCS 编译时配置选项
- 合并了对 RTCS 源程序所做的修改

6.2 前言

在重新编译 RTCS 前，建议:

- 阅读关于 MQX RTOS 重编译指令的文档《MQX RTOS 用户指南》。类似概念同样适用于 RTCS。
- 阅读 Freescale MQX RTOS 随附的《MQX RTOS 版本说明》，以获取特定于您的目标环境和硬件的信息。
- 具备目标环境所需的工具:
  - 编译器
  - 汇编器
  - 链接器
- 熟悉 RTCS 的目录结构和重编译指令 (版本说明文档中对此均有描述)，以及后续章节中介绍的指令。

6.3 Freescale MQX RTOS 中的 RTCS 编译工程

RTCS 编译工程的结构与 Freescale MQX RTOS 中包含的其他内核库工程十分相似。对于给定的开发环境（例如 CodeWarrior），编译工程位于 rtcs\build\<compiler>目录中。虽然 RTCS 代码并非特定于某些开发板或处理器衍生产品，但每个支持的开发板都有一个独立的 RTCS 编译工程。此外，结果库文件会编译输出到特定板的输出目录 lib\<board>.<compiler>。

之所以要将独立于开发板的代码编译到特定开发板的输出目录，原因在于这样更便于分别配置每块开发板。编译时用户配置文件取自特定开发板目录 config\<board>。换言之，对于两块不同的开发板，用户可能希望编译不同的结果库代码。

有关用户配置文件或如何创建自定义配置和编译工程的更多详情，请参见《MQX RTOS 用户指南》。

6.3.1 编译后处理

RTCS 编译工程还设定了执行编译后的批处理文件，将所有公共头文件复制到目标目录中。这使得输出\lib 目录成为应用代码可访问的唯一位置。MQX RTOS 应用工程需要使用 RTCS 服务，无需引用 RTCS 源码树。

6.3.2 编译目标

CodeWarrior 开发环境允许使用多种编译配置 (也被称为编译目标)。Freescale MQX RTOS RTCS 中的所有工程至少包含两个编译目标:

- 调试目标 — 编译器优化设为低，以简化调试。
- 发布目标 — 编译器优化设为最高，以实现最小的代码大小和最快的执行速度。这样得到的代码很难进行调试。

6.4 重新编译 Freescale MQX RTCS

重新编译 MQX RTOS RTCS 库非常简单，只需在开发环境中打开合适的编译工程，并对其进行编译。切记选择合适的编译目标进行编译或编译所有目标。

关于重新编译 MQX RTOS RTCS 的具体信息和示例应用，请参见 Freescale MQX RTOS 发行版随附的版本记录。

第 7 章
函数参考

7.1 函数列表格式

这是函数、编译器内部对象或宏常用的条目格式。

7.1.1 function_name()

简短描述函数 function_name()的功能。

概要

为函数 function_name()提供一个原型。

```
<return_type> function_name(
       <type_1> parameter_1,
       <type_2> parameter_2,
       ...
       <type_n> parameter_n)
```

参数

parameter_1 [in] — 指向 x 的指针

parameter_2 [out] — y 的句柄

parameter_n [in/out] — 指向 z 的指针

参数传递分类如下:

- *In* — 表示该函数使用您向其提供的参数中的一个或多个值，不保存任何更改。
- 输出
- *Out* — 表示该函数保存您向其提供的参数中的一个或多个值。您可以检查保存的值，以找到关于应用的有用信息。

- 输入/输出
- *In/out* — 表示该函数更改您向其提供的参数中的一个或多个值，并保存结果。您可以检查保存的值，以找到关于应用的有用信息。

说明

描述函数 function_name()。此部分还描述可能适用的任何特征或限制:

- 在某些特定情况下，函数会阻塞，或可能阻塞。
- 函数必须作为任务启动。
- 函数创建一个任务。
- 函数具有一些可能不太明显的先决条件。
- 函数有限制条件或特殊行为。

返回值

指明由函数 function_name()返回的任何值。

另请参见

列出与函数 function_name()相关的其他函数或数据类型。

示例

提供一个示例或对示例的引用，用以说明函数 function_name()的用途。

函数列表

此部分按照字母顺序提供函数列表。

7.2 accept()

创建一个新的流套接字，以接受从远程端点发来的连接请求。

概要

```
uint32_t accept(
        uint32_t        socket,
        sockaddr    *   peeraddr,
        uint16_t    *   addrlen)
```

参数

socket [in] — 用于父流套接字的句柄。

peeraddr [out] — 指针，指向放置远程端点标识符的位置。

addrlen [in/out] — 当传入时：为指针，指向位置 peeraddr 所指内容的字节长度。当传出时：为远程端点标识符的完整大小，以字节为单位。

说明

该函数通过为传入连接创建新的流套接字，以接受传入连接。父套接字（socket）必须处于监听状态，并且在每次创建新的套接字后保持监听。

通过 accept()创建的新套接字继承了监听套接字的链接层选项。新套接字与父套接字具有相同的本地端点和套接字选项。远程端点是连接的发起者。

该函数保持阻塞，直到出现可用的传入连接。如果父套接字（监听）由另一任务中调用 closesocket()而关闭，且 accept()阻塞时，accept()返回-1（RTCS_SOCKET_ERROR），而 RTCS_errno 设为 RTCSERR_SOCK_CLOSED。如果父套接字（监听）由另一任务中调用 shutdownsocket()而关闭，accept()返回-1（RTCS_SOCKET_ERROR），而 RTCS_errno 设为 RTCSERR_SOCK_ESHUTDOWN。

返回值

- 新的流套接字的句柄（成功）
- RTCS_SOCKET_ERROR（失败）

另请参见

- bind()
- connect()
- listen()
- socket()

示例

a) 套接字接受 IPv4 连接。

```
uint32_t    handle;
uint32_t    child_handle;
sockaddr    remote_sin;
uint16_t    remote_addrlen;
uint32_t    status;
...
status = listen(handle, 0);
if (status != RTCS_OK)
{
    printf("\nError, listen() failed with error code %lx", status);
}
else
```

```
{
    remote_addrlen = sizeof(remote_sin);
    child_handle = accept(handle, &remote_sin, &remote_addrlen);
    if (child_handle != RTCS_SOCKET_ERROR)
    {
        printf("\nConnection accepted from %lx, port %d",
        remote_sin.sin_addr, remote_sin.sin_port);
    }
    else
    {
        uint32_t rtcserrno = RTCS_get_errno();
        /* The value will be RTCSERR_SOCK_CLOSED if closesocket()
         * has been called (from other task).
         * After closesocket(), the socket cannot be used.
         */
        if (rtcserrno == RTCSERR_SOCK_CLOSED)
        {
            service_closed_listensock();
        }
        else
        {
            status = RTCS_geterror(handle);
            printf("Error, accept() failed with error code %lx",status);
        }
    }
}
```

b) 套接字接受端口 7007 上的 IPv6 连接。

```
uint32_t    sock, sock6;
sockaddr_in6 laddr6, raddr6;
uint16_t rlen;
memset(&laddr6,0x0,sizeof(laddr6));
laddr6.sin6_port = 7007;
laddr6.sin6_family = AF_INET6;
laddr6.sin6_addr = in6addr_any;
laddr6.sin6_scope_id = 0;
sock6 = socket(AF_INET6, SOCK_STREAM, 0);
if(RTCS_SOCKET_ERROR == sock6)
{
   printf("Error, socket() failed\n");
   _task_block();
}
error = bind(sock6, &laddr6, sizeof(laddr6));
if(RTCS_OK != error)
{
   printf("bind() failed, error 0x%lx\n", error);
   _task_block();
}
error = listen(sock6, 0);
if(RTCS_OK != error)
{
  printf("listen() failed - 0x%lx\n", error);
  _task_block();
}
sock = RTCS_selectset(&sock6,1,0);
if(RTCS_SOCKET_ERROR == sock)
{
  printf("selectset() failed - 0x%lx\n", RTCS_geterror(sock6));
  _task_block();

}
if(sock == sock6)
{
    rlen = sizeof(raddr6);
    sock = accept(sock6, &raddr6, &rlen);
    if(RTCS_SOCKET_ERROR == sock)
    {
        uint32_t rtcserrno = RTCS_get_errno();
```

```
        /* The value will be RTCSERR_SOCK_CLOSED if closesocket()
         * has been called (from other task).
         * After closesocket(), the socket cannot be used.
         */
        if (rtcserrno == RTCSERR_SOCK_CLOSED)
        {
            service_closed_listensock();
        }
    }
    else
    {
        status = RTCS_geterror(sock6);
        printf("Error, accept() failed with error code %lx",status);
        _task_block();
    }
}
```

7.3 ARP_stats()

获取指向 RTCS 为接口收集的 ARP 统计数据的指针。

概要

```
ARP_STATS_PTR ARP_stats(
        _rtcs_if_handle rtcs_if_handle)
```

参数

rtcs_if_handle [in] — 从 RTCS_if_add()获得的 RTCS 接口句柄。

返回值

- 指针，指向用于 rtcs_if_handle 的 ARP_STATS 结构（成功）。
- 零（失败：rtcs_if_handl*e* 无效）。

另请参见

- ENET_get_stats()
- ICMP_STATS
- inet_pton()
- IPIF_stats()
- RTCS_if_add()
- TCP_stats()
- UDP_stats()
- ARP_STATS

示例

使用 RTCS 统计函数以显示所收数据包的统计数据。

```
void display_rx_stats(void)
{
  IP_STATS_PTR    ip;
  IGMP_STATS_PTR  igmp;
  IPIF_STATS      ipif;
  ICMP_STATS_PTR  icmp;
  UDP_STATS_PTR   udp;
  TCP_STATS_PTR   tcp;
  ARP_STATS_PTR   arp;
  _rtcs_if_handle ihandle;
  _enet_handle    ehandle;

  ENET_initialize(ENET_DEVICE, enet_local, 0, &ehandle);
  RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);

  ip = IP_stats();
  igmp = IGMP_stats();
  ipif = IPIF_stats(ihandle);
  icmp = ICMP_stats();
  udp = UDP_stats();
  tcp = TCP_stats();
  arp = ARP_stats(ihandle);
  printf("\n%d IP packets received",   ip->ST_RX_TOTAL);
  printf("\n%d IGMP packets received", igmp->ST_RX_TOTAL);
  printf("\n%d IPIF packets received", ipif->ST_RX_TOTAL);
  printf("\n%d TCP packets received",  tcp->ST_RX_TOTAL);
  printf("\n%d UDP packets received",  udp->ST_RX_TOTAL);
  printf("\n%d ICMP packets received", icmp->ST_RX_TOTAL);
  printf("\n%d ARP packets received",  arp->ST_RX_TOTAL);
}
```

7.4 bind()

将本地地址绑定到套接字

概要

```
uint32_t bind(
        uint32_t      socket,
        sockaddr    * localaddr,
        uint16_t        addrlen)
```

参数

socket [in] — 用于待绑定的套接字的套接字句柄。

localaddr [in] — 指针，指向绑定套接字的本地端点标识符（参见说明）。

addrlen [in] — localaddr 所指向的内容的字节长度。

说明

需要下列 localaddr 输入值:

sockaddr 字段	需要的输入值
sin_family	AF_INET
sin_port	以下之一: • 套接字的本地端口号。 • 零（若要确定 RTCS 所选的端口号，调用 getsockname()。）
sin_addr	以下之一: • 之前通过调用 RTCS_if_bind 函数绑定的 IP 地址。 • *INADDR_ANY*。

sockaddr 字段	需要的输入值
sin6_family	AF_INET6
sin6_port	以下之一: • 套接字的本地端口号。 • 零（若要确定 RTCS 所选的端口号，调用 getsockname()。）
sin6_addr	IPv6 地址。
sin6_scope_id	作用域索引。

TCP/IP 服务器通常绑定 INADDR_ANY，因此一个服务器实例可以服务所有 IP 地址。

该函数阻塞，但是 RTCS 会立即执行命令，并被套接字层回复。

返回值

- RTCS_OK（成功）
- 特定错误代码（失败）

另请参见

- RTCS_if_bind 函数系列
- socket()
- sockaddr_in
- sockaddr

示例

a) 将一个套接字绑定到端口号 2010。

```
uint32_t sock;
sockaddr_in local_sin;
uint32_t result;
...
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock == RTCS_SOCKET_ERROR)
{
  printf("\nError, socket create failed");
  return;
}
memset((char *) &local_sin, 0, sizeof(local_sin));
local_sin.sin_family = AF_INET;
local_sin.sin_port = 2010;
local_sin.sin_addr.s_addr = INADDR_ANY;
result = bind(sock, (struct sockaddr *)&local_sin, sizeof (sockaddr_in));
if (status != RTCS_OK)
  printf("\nError, bind() failed with error code %lx", result);
```

b) 将一个套接字绑定到端口号 7007，使用 IPv6 协议。

```
uint32_t sock, sock6;
sockaddr_in6 laddr6, raddr6;
uint16_t rlen;
memset(&laddr6,0x0,sizeof(laddr6));
laddr6.sin6_port = 7007;
laddr6.sin6_family = AF_INET6;
laddr6.sin6_addr = in6addr_any;
laddr6.sin6_scope_id = 0;
sock6 = socket(AF_INET6, SOCK_STREAM, 0);
if(RTCS_SOCKET_ERROR == sock6)
{
  printf("Error, socket() failed\n");
  _task_block();
}
error = bind(sock6, &laddr6, sizeof(laddr6));
if(RTCS_OK != error)
{
  printf("bind() failed, error 0x%lx\n", error);
  _task_block();
}
```

7.5 closesocket()

关闭套接字。closesocket()返回后，应用不能再使用该套接字。

概要

```
int32_t closesocket(uint32_t sock)
```

参数

- *sock [in]* – 套接字句柄

说明

数据报套接字立即关闭，未完成的 recvfrom()调用立即返回，排队的传入数据包被丢弃。未连接的流套接字立即关闭。已连接流的行为取决于 SO_LINGER 套接字选项：

表 7-1. 已连接流 closesocket()的行为基于 SO_LINGER 套接字选项

SO_LINGER	（默认）			
l_onoff	0	1	1	1
l_linger_ms	无关位	0	>0 毫秒	>0 毫秒
4 路正常关闭			在 l_linger_ms 超时限超过前完成。	未在 l_linger_ms 时限内完成。
关闭类型	正常关闭	强制关闭（重置）	正常关闭	强制关闭（重置）
返回的时机	立即（无阻塞）	立即（无阻塞）	当 4 路正常关闭完成后（阻塞）	当 l_linger_ms 到期后（阻塞）。

如果在一个套接字上调用 closesocket()函数，则对 send()和 recv()的函数调用立即返回。

若延迟等待功能关闭 (l_onoff = 0)，TIMEWAIT_TIMEOUT 定时器启动，并通过应用层调用 closesocket()函数。如果在 TIMEWAIT_TIMEOUT 时限内未完成正常的连接关闭, 用于套接字的 TCB 将会关闭并向远程主机发送重置信号。若延迟等待功能未关闭(l_onoff != 0)，l_linger_ms 定时器启动，并通过应用层调用 closesocket()函数。如果在 l_linger_ms 时限内未完成正常的连接关闭，用于套接字的 TCB 将会关闭并向远程主机发送重置信号。如果 l_linger_ms 为零，则 (TCB 关闭和重置) 会立即发生。如果 RTCS 首先发送 FIN，用于套接字的 TCB 在连接被正常关闭时保持 TIMEWAIT_TIMEOUT 时间 (TCP 连接处于 TIME_WAIT 状态)。

默认情况下，RTCS 将 TIMEWAIT_TIMEOUT 设置为 2 秒 (DEFAULT_TIMEWAIT_TIMEOUT 编译时间宏)。对于要求 TCP 连接处于 TIME_WAIT 状态以保持更长时间的应用，使用套接字选项 TIMEWAIT_TIMEOUT。

映射 RTCS 的旧 API, (shutdown() /RTCS_shutdown()): shutdown(FLAG_CLOSE_TX) 或 shutdown(0)对应于正常关闭过程，而 shutdown(FLAG_ABORT_CONNECTION) 对应于强制 (重置) 关闭。

返回值

- 零 (成功 - RTCS_OK)
- 非零 (失败 - 特定错误代码)

另请参见

- shutdownsocket()
- setsockopt()

示例

```
closesocket(clientsock); /* default Non-blocking graceful close */
```

7.6 connect()

将流套接字连接到远程端点，或为数据报套接字设定一个远程端点。

概要

```
uint32_t connect(
         uint32_t                socket,
         sockaddr      *         destaddr,
         uint16_t                  addrlen)
```

参数

socket [in] — 用于待连接的流套接字的句柄。

destaddr [in] — 指向远程端点标识符的指针。

addrlen [in] — *destaddr* 所指内容的字节长度。

说明

connect()函数可能会使用多次。每当 connect()被调用时，当前端点就被新的端点替代。

如果 connect()失败，套接字将保留绑定状态，或没有远程端点。

若同流套接字一起使用，函数在远程端点处于以下情况时会失败:

- 拒绝连接请求（可能会立即发生）。
- 不可到达（导致连接超时）。

如果函数成功，应用可使用该套接字来传输数据。

若同数据报套接字一起使用，该函数具有如下效果:

- 可使用 send()函数代替 sendto()将一个数据报发送到 destaddr。
- sendto()的行为未改变：仍然可用于将数据报发送到任何对等端。
- 套接字仅从 destaddr 接收数据报。

该任务保持阻塞，直到连接被接受或连接超时套接字选项到期。

返回值

- RTCS_OK（成功）
- 特定错误代码（失败）

另请参见

- accept()
- bind()
- getsockopt()
- listen()
- setsockopt()
- socket()

示例：数据流套接字

a) 该连接使用 IPv4 协议。

```
uint32_t              sock;
uint32_t              child_handle;
sockaddr_in          remote_sin;
uint16_t              remote_addrlen = sizeof(sockaddr_in);
uint32_t              result;
...

/* Connect to 192.203.0.83, port 2011: */
memset((char *) &remote_sin, 0, sizeof(sockaddr_in));
remote_sin.sin_family       = AF_INET;
remote_sin.sin_port         = 2011;
remote_sin.sin_addr.s_addr  = 0xC0A80001; /* 192.168.0.1 */

result = connect(sock, (struct sockaddr *)&remote_sin, remote_addrlen);

if (result != RTCS_OK)
{
   printf("\nError--connect() failed with error code %lx.",
                                                              result);
}  else {
   printf("\nConnected to %lx, port %d.",
          remote_sin.sin_addr.s_addr, remote_sin.sin_port);
}
```

b) 该连接使用 IPv6 协议。

```
 struct addrinfo     hints;        /* Used for getaddrinfo()*/
 struct addrinfo    *addrinfo_res; /* Used for getaddrinfo()*/
 uint32_t sock;
 uint32_t error;

/* Extract IP address and detect family, here we will get scope_id too. */
memset(&hints,0,sizeof(hints));
hints.ai_family      = AF_UNSPEC; /* Allow IPv4 or IPv6 */
hints.ai_socktype    = SOCK_STREAM;
if (getaddrinfo("fe80::e5ec:43fc:4aca:bf13","7007", &hints, &addrinfo_res) != 0)
{
  printf("GETADDRINFO error\n");
  /* We can return right here and do not need free freeaddrinfo(addrinfo_res)*/
```

```
    return SHELL_EXIT_ERROR;
  }

  sock = socket(addrinfo_res->ai_family, SOCK_STREAM, 0);
  if(RTCS_SOCKET_ERROR == sock)
  {
    printf("Socket create failed\n");
    freeaddrinfo(addrinfo_res);
    return;
  }

  error = connect(sock, addrinfo_res->ai_addr, addrinfo_res->ai_addrlen);
  if(RTCS_OK != error)
  {

    printf("Connect failed, return code 0x%lx\n", error);
    freeaddrinfo(addrinfo_res);
    return;
  }

  freeaddrinfo(addrinfo_res);
```

7.7 DHCP_find_option()

针对特定选项类型搜索 DHCP 消息。

概要

```
unsigned char       *DHCP_find_option(
  unsigned char     *msgptr,
  uint32_t          msglen,
  uchar             option)
```

参数

msgptr [in/out] — 指向 DHCP 消息的指针。

msglen [in] — 消息字节数。

option [in] — 要搜索的选项类型 （参见 RFC 2131）。

说明

msgptr 指针指向 DHCP 消息中的选项，其格式根据 RFC 2131 和 RFC 2132 设置。应用负责解析选项和读取值。

返回的指针必须传递给宏 ntohl 或宏 ntohs，以提取选项的值。这些宏可将值转换成按照主机的字节顺序排列。

返回值

- 指向 DHCP 消息中特定选项的指针，按照网络字节排列 （成功）。
- 零 （不存在特定类型的选项）。

另请参见

- DHCPCLNT_find_option()

示例

```
/* Get a pointer to the start of the DHCP server's name from a
   packet (like a DH_OFFER packet) recieved from the server */

uchar * buffer_ptr; /* This is a DHCP packet recieved
                          from a server */
uint32_t buffer_size;
uchar * optptr;
optptr = DHCPCLNT_find_option(buffer_ptr, buffer_size, DHCPOPT_SERVERNAME);
```

7.8 DHCP_option_addr()

向 DHCP 服务器的 DHCP 选项列表中添加 IP 地址。

概要

```
bool    DHCP_option_addr(
           unsigned char * *optptr,
           uint32_t      *  optlen,
           uchar           opttype,
           _ip_address      optval)
```

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针:

在 optval 添加前传入。

在 optval 添加后传出。

opttype [in] — 要添加到列表中的选项类型（参见 RFC 2132）。

optval [in] — 要添加的 IP 地址。

说明

函数 DHCP_option_addr()向 DHCP 服务器的 DHCP 选项列表中添加 IP 地址。随后，应用将参数 optptr（指向选项列表的指针）传递给 DHCPSRV_ippool_add()。

返回值

- TRUE（成功）
- FALSE（失败：选项列表中空间不足）

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_string()
- DHCP_option_variable()

示例

请参见 DHCPSRV_init() 。

7.9 DHCP_option_addrlist()

向 DHCP 服务器的 DHCP 选项列表中添加 IP 地址列表

概要

```
bool     DHCP_option_addrlist(
            unsigned char    *        *optptr,
            uint32_t        *          optlen,
            uchar                      opttype,
            _ip_address      *         optval,
            uint32_t                   listlen)
```

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针:

在 optval 添加前传入。

在 optval 添加后传出。

opttype [in] — 要添加到列表中的选项类型 (参见 RFC 2132)。

optval [in] — 指向 IP 地址列表的指针。

listlen [in] — 列表中 IP 地址的个数。

说明

函数 DHCP_option_addrlist()向 DHCP 服务器的 DHCP 选项列表中添加 optval 引用的 IP 地址列表。随后，应用将参数 optptr 或指向选项列表的指针传递给 DHCPSRV_ippool_add()。

返回值

- TRUE（成功）
- FALSE（失败：选项列表中空间不足）

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()
- DHCP_option_addr()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_string()
- DHCP_option_variable()

示例

请参见 DHCPSRV_init() 。

7.10 DHCP_option_int16()

向 DHCP 服务器的 DHCP 选项列表中添加一个 16 位值。

概要

```
bool DHCP_option_int16(
  unsigned char  *  *optptr,
  uint32_t       * optlen,
uchar             opttype,
  uint16_t        optval)
```

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针：

在 *optval* 添加前传入。

在 *optval* 添加后传出。

opttype [in] — 要添加到列表中的选项类型（参见 RFC 2132）。

optval [in] — 要添加的值。

说明

函数 DHCP_option_int16()向 DHCP 服务器的 DHCP 选项列表中添加一个 16 位值 optval。随后，应用将参数 optptr 或指向选项列表的指针传递给 DHCPSRV_ippool_add()。

返回值

- TRUE（成功）
- FALSE（失败：选项列表中空间不足）

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()
- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_string()
- DHCP_option_variable()

示例

请参见 DHCPSRV_init() 。

7.11 DHCP_option_int32()

向 DHCP 服务器的 DHCP 选项列表中添加一个 32 位值。

概要

```
bool DHCP_option_int32(
   unsigned char * *optptr,
   uint32_t     *  optlen,
   uchar           opttype,
   uint32_t          optval)
```

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针:

在 optval 添加前传入。

在 optval 添加后传出。

opttype [in] — 要添加到列表中的选项类型（参见 RFC 2132)。

optval [in] — 要添加的值。

说明

函数 DHCP_option_int32()向 DHCP 服务器的 DHCP 选项列表中添加一个 32 位值。随后，应用将参数 optptr 或指向选项列表的指针传递给 DHCPSRV_ippool_add()。

返回值

- TRUE（成功)
- FALSE（失败：选项列表中空间不足)

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()
- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_string()
- DHCP_option_variable()

示例

请参见 RTCS_if_bind_DHCP() 和 DHCPSRV_init() 。

7.12 DHCP_option_int8()

向 DHCP 服务器的 DHCP 选项列表中添加一个 8 位值。

概要

```
bool DHCP_option_int8(
  unsigned char * *optptr,
  uint32_t *      optlen,
  uchar           opttype,
  uchar           optval)
```

说明

函数 DHCP_option_int8()向 DHCP 服务器的 DHCP 选项列表中添加一个 8 位值。随后，应用将参数 optptr 或指向选项列表的指针传递给 DHCPSRV_ippool_add()。

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针:

在 optval 添加前传入。

在 optval 添加后传出。

opttype [in] — 要添加到列表中的选项类型（参见 RFC 2132）。

optval [in] — 要添加的值。

返回值

- TRUE（成功）
- FALSE（失败：选项列表中空间不足）

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()
- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int16()
- DHCP_option_int32()

- DHCP_option_string()
- DHCP_option_variable()

示例

请参见 DHCPSRV_init() 。

7.13 DHCP_option_string()

向 DHCP 服务器的 DHCP 选项列表中添加一个字符串。

概要

```
uint32_t DHCP_option_string(
         unsigned char *   *optptr,
         uint32_t *         optlen,
         uchar              opttype,
         char              *optval)
```

说明

函数 DHCP_option_string()向 DHCP 服务器的 DHCP 选项列表中添加一个字符串。随后，应用将参数 optptr 或指向选项列表的指针传递给 DHCPSRV_ippool_add()。

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针:

在 optval 添加前传入。

在 optval 添加后传出。

opttype [in] — 要添加到列表中的选项类型（参见 RFC 2132)。

optval [in] — 要添加的字符串。

返回值

- TRUE（成功)
- FALSE（失败：选项列表中空间不足)

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()

- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_variable()

示例

请参见 DHCPSRV_init() 。

7.14 DHCP_option_variable()

向 DHCP 服务器的 DHCP 选项列表中添加一个变长选项。

概要

```
uint32_t DHCP_option_variable(
         unsigned char * *optptr,
         uint32_t      * optlen,
         uchar           opttype,
         uchar         * optdata,
         uint32_t         datalen)
```

参数

optptr [in/out] — 指向选项列表的指针。

optlen [in/out] — 指向选项列表中剩余字节数的指针:

在 optval 添加前传入。

在 optval 添加后传出。

opttype [in] — 要添加到列表中的选项类型 (参见 RFC 2132)。

optdata [in] — 要添加的字节序列。

optlen [in] — *optptr* 指向的字节数。

说明

函数 DHCP_option_variable()向 DHCP 服务器的 DHCP 选项列表中添加一个变长选项。使用此函数创建您要传给 DHCPSRV_ippool_add()和 RTCS_if_bind_DHCP()的 optptr 缓冲区。

返回值

- TRUE（成功）
- FALSE（失败）

另请参见

- DHCPCLNT_find_option()
- DHCPSRV_ippool_add()
- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_string()
- RTCS_if_bind_DHCP()

示例

请参见 RTCS_if_bind_DHCP() 。

7.15 DHCPCLN6_init()

该函数可启动 DHCPv6 客户端。

概要

```
uint32_t DHCPCLN6_init(
DHCPCLN6_PARAM_STRUCT *params);
```

参数

params [in] - 客户端参数

说明

使用该函数启动 DHCPv6 客户端。该函数保持阻塞，直到启动完成。

返回值

- 如果初始化成功，返回值为客户端句柄，否则为零。

另请参见

- DHCPCLN6_release()
- DHCPCLN6_PARAM_STRUCT

示例

```
/*
* Start DHCPv6 client on interface no.0 with link checking enabled. Wait
* 5 seconds and then stop it.
*/
uint32_t              handle;
char                  *result_s;
DHCPCLN6_PARAM_STRUCT params = {0};
uint32_t               result;

params.flags |= DHCPCLN6_FLAG_CHECK_LINK;
params.interface = RTCS_if_get_handle(0);
handle = DHCPCLN6_init(params);
fprintf(stdout, "DHCPv6 initialization %s.\n", (handle == 0) ? "failed" : "successful");
if (handle != 0)
{
   uint32_t i;
   /* Wait 15 seconds for address. */
   for(i = 0; i < 15; i++)
   {
       if (DHCPCLN6_get_status(dhcp6_handle) == DHCPCLN6_STATUS_BOUND)
       {
          printf("Address from DHCPv6 server obtained.\n");
          break;
       }
       _time_delay(1000);
   }

   if (i == 15)
   {
       printf("Failed to obtain address from DHCPv6 server!\n");
   }
   _time_delay(5000);
   result = DHCPCLN6_release(handle);
   fprintf(stdout, "DHCPv6 release %s.\n", (result == RTCS_OK) ? "successful" : "failed");
}
```

7.16 DHCPCLN6_release()

该函数可停止 DHCPv6 客户端。

概要

```
uint32_t DHCPCLN6_release(
uint32_t handle);
```

参数

handle [in] - DHCPv6 客户端的句柄，由函数 DHCPCLN6_init() 创建。

说明

使用该函数停止 DHCPv6 客户端。使用该函数后能够释放客户端获得的所有地址。该函数保持阻塞，直到释放完成。

返回值

- 如果释放成功，返回值为 RTCS_OK ，否则为 RTCS_ERROR 。

示例

- 参见关于 DHCPCLN6_init() 的示例。

另请参见

- DHCPCLN6_init()

7.17 DHCPCLN6_get_status()

该函数在应用层代码需要查看客户端状态时被调用。

概要

```
DHCPCLN6_STATUS DHCPCLN6_get_status(
uint32_t handle);
```

参数

handle [in] - DHCPv6 客户端的句柄，由函数 DHCPCLN6_init() 创建。

说明

使用该函数读取 DHCPv6 客户端的当前状态。返回值表示客户端是否分配了某些地址，以及客户端是否正在运行。

返回值

- 从 DHCPCLN6_STATUS enum 读取的客户端状态。

示例

- 参见关于 DHCPCLN6_init() 的示例。

另请参见

- DHCPCLN6_init()
- DHCPCLN6_STATUS

7.18 DHCPCLNT_find_option()

针对特定选项类型搜索 DHCP 消息。

概要

```
unsigned char *DHCPCLNT_find_option(
        unsigned char *msgptr,
        uint32_t      msglen,
        uchar         option)
```

参数

msgptr [in/out] — 指向 DHCP 消息的指针。

msglen [in] — 消息字节数。

option [in] — 要搜索的选项类型 （参见 RFC 2131)。

说明

msgptr 指针指向 DHCP 消息中的选项, 其格式根据 RFC 2131 和 RFC 2132 设置。应用负责解析选项和读取值。

返回的指针必须传递给宏 ntohl 或宏 ntohs， 以提取选项的值。这些宏可将值转换成按照主机的字节顺序排列。

返回值

- 指向 DHCP 消息中特定选项的指针， 按照网络字节顺序排列 （成功)。
- 零 （不存在特定类型的选项)。

另请参见

- DHCP_find_option()

7.19 DHCPCLNT_release()

释放不再需要使用的 DHCP 客户端。

概要

```
unsigned char *DHCPCLNT_release(
        _rtcs_if_handle handle)
```

参数

handle [in] — 指针，指向不再需要使用的接口。

说明

当您的应用不再需要某个 DHCP 客户端时，可使用函数 DHCPCLNT_release()将其释放。

DHCPCLNT_release()的功能:

- 取消 DHCP 状态机中的定时器事件。
- 将状态设为释放，使具有该状态的资源释放。
- 对接口进行解绑。
- 停止监听 DHCP 端口。
- 释放资源。

返回值

- 空（成功）
- 错误代码（失败）

另请参见

- RTCS_if_bind_DHCP()

示例

```
_rtcs_if_handle ihandle;
/* start RTCS task, add an interface and bind it with
   RTCS_if_bind_DHCP */
/* do some stuff with the interface */
/* all done */
DHCPCLNT_release(ihandle);
```

7.20 DHCPSRV_init()

启动 DHCP 服务器。

概要

```
uint32_t DHCPSRV_init(
       char *name,
```

```
              uint32_t  priority,
              uint32_t  stacksize)
```

参数

name [in] — 服务器任务的名称。

priority [in] — 服务器任务的优先级。

stacksize [in] — 服务器任务的协议栈大小。

说明

函数 DHCPSRV_init()可启动 DHCP 服务器并创建 DHCPSRV_task。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- DHCPCLNT_find_option()
- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_string()
- DHCP_option_variable()

示例

启动 DHCP 服务器并设置其选项:

```
DHCPSRV_DATA_STRUCT          dhcpsrv_data;
uchar                        dhcpsrv_options[200];
_ip_address                 routers[3];
unsigned char               *optptr;
uint32_t                    optlen;
uint32_t                    error;
/* Start DHCP Server: */
error = DHCPSRV_init("DHCP server", 7, 2000);
if (error != RTCS_OK) {
   printf("\nFailed to initialize DHCP Server, error %x", error);
   return;
}
printf("\nDHCP Server running");
```

```
/* Fill in the required parameters: */
/* 192.168.0.1: */
dhcpsrv_data.SERVERID = 0xC0A80001;
/* Infinite leases: */
dhcpsrv_data.LEASE = 0xFFFFFFFF;
/* 255.255.255.0: */
dhcpsrv_data.MASK = 0xFFFFFF00;
/* TFTP server address: */
dhcpsrv_data.SADDR = 0xC0A80002;
memset(dhcpsrv_data.SNAME, 0, sizeof(dhcpsrv_data.SNAME));
memset(dhcpsrv_data.FILE, 0, sizeof(dhcpsrv_data.FILE));
/* Fill in the options: */
optptr = dhcpsrv_options;
optlen = sizeof(dhcpsrv_options);
/* Default IP TTL: */
DHCPSRV_option_int8(&optptr, &optlen, 23, 64);
/* MTU: */
DHCPSRV_option_int16(&optptr, &optlen, 26, 1500);
/* Renewal time: */
DHCPSRV_option_int32(&optptr, &optlen, 58, 3600);
/* Rebinding time: */
DHCPSRV_option_int32(&optptr, &optlen, 59, 5400);
/* Domain name: */
DHCPSRV_option_string(&optptr, &optlen, 15, "arc.com");
/* Broadcast address: */
DHCPSRV_option_addr(&optptr, &optlen, 28, 0xC0A800FF);
/* Router list: */
routers[0] = 0xC0A80004;
routers[1] = 0xC0A80005;
routers[2] = 0xC0A80006;
DHCPSRV_option_addrlist( &optptr, &optlen, 3, routers, 3);
/* Serve addresses 192.168.0.129 to 192.168.0.135 inclusive: */
DHCPSRV_ippool_add(0xC0A80081, 7, &dhcpsrv_data, dhcpsrv_options,
                   optptr - dhcpsrv_options);
```

7.21 DHCPSRV_ippool_add()

向 DHCP 服务器提供要服务的 IP 地址块。

概要

```
uint32_t DHCPSRV_ippool_add(
         _ip_address              ipstart,
         uint32_t                 ipnum,
         DHCPSRV_DATA_STRUCT_PTR  params_ptr,
         unsigned char            *optptr,
         uint32_t                 optlen)
```

参数

ipstart [in] — 提供的首个 IP 地址。

ipnum [in] — 提供的 IP 地址的个数。

params_ptr [in] — 指针，指向与 IP 地址相关的配置信息。

optptr [in] — 指针，指向与 IP 地址相关的可选配置信息。

optlen [in] — optptr 指向的字节数。

说明

函数 DHCPSRV_ippool_add()向 DHCP 服务器提供要服务的 IP 地址块。DHCP 服务器任务必须在您调用此函数之前创建（通过调用 DHCPSRV_init()）。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- DHCPCLNT_find_option()
- DHCP_option_addr()
- DHCP_option_addrlist()
- DHCP_option_int8()
- DHCP_option_int16()
- DHCP_option_int32()
- DHCP_option_string()
- DHCP_option_variable()
- DHCPSRV_init()
- DHCP_DATA_STRUCT

示例

请参见 DHCPSRV_init()

7.22 DHCPSRV_set_config_flag_off()

禁用地址探测。

概要

```
uint32_t DHCPSRV_set_config_flag_off (
            uint32_t                    flag)
```

参数

flag [in] — DHCP 服务器地址探测标志

说明

默认情况下，RTCS DHCP 服务器在向客户端发布地址之前会先对所请求 IP 地址的网络进行探测。如果服务器接收到响应，则其将发送一个 NAK 回复，并等待客户端请求新的地址。您可以禁用探测,以减少时间和流量上的开销。若要实现该操作，将 DHCPSVR_FLAG_DO_PROBE 标志传递给 DHCPSRV_set_config_flag_off()。

该函数可在 DHCPSRV_init()之后的任何时候调用。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- DHCPSRV_set_config_flag_on()

示例

```
#define DHCP_DO_PROBING 1
int dhcp_do_probing = DHCP_DO_PROBING;
/*init*/
/*setup*/
if (dhcp_do_probing) {
    DHCPSRV_set_config_flag_on(DHCPSVR_FLAG_DO_PROBE);
    }
else {
    DHCPSRV_set_config_flag_off(DHCPSVR_FLAG_DO_PROBE);
    }
```

7.23 DHCPSRV_set_config_flag_on()

重新启用地址探测

概要

```
    uint32_t DHCPSRV_set_config_flag_on (
             uint32_t                        flag
```

参数

flag [in] — DHCP 服务器地址探测标志

说明

默认情况下，RTCS DHCP 服务器在向客户端发布地址之前会先对所请求 IP 地址的网络进行探测。如果服务器接收到响应，则其将发送一个 NAK 回复，并等待客户端请求新的地址。如果您之前已禁用探测，将 DHCPSVR_FLAG_DO_PROBE 标志传递给 DHCPSRV_set_config_flag_on()以重新启用探测。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

示例

```
#define DHCP_DO_PROBING 1
int dhcp_do_probing = DHCP_DO_PROBING;
/*init*/
/*setup*/
if (dhcp_do_probing) {
   DHCPSRV_set_config_flag_on(DHCPSVR_FLAG_DO_PROBE);
   }
else {
   DHCPSRV_set_config_flag_off(DHCPSVR_FLAG_DO_PROBE);
   }
```

7.24 ECHOCLN_connect

连接到 RFC 862 ECHO 服务器。

概要

```
uint32_t     ECHOCLN_connect (const struct addrinfo *addrinfo_ptr)
```

参数

- *addrinfo_ptr [in]* – 指针，指向远程主机的 addrinfo 结构体

说明

尝试连接到远程主机上的服务器。远程主机由 addrinfo 结构体指定，该结构体可通过 getaddrinfo()获取。

返回值

- RTCS_SOCKET_ERROR（失败）
- 套接字句柄（成功）。

另请参见

- ECHOCLN_process
- getaddrinfo()

示例

```
int32_t i_result;
```

```
uint32_t sock = RTCS_SOCKET_ERROR;
struct addrinfo *result = NULL,
 *ptr = NULL,
 hints;

memset( &hints, 0, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

i_result = getaddrinfo("192.168.1.202", "7", &hints, &result);
if ( i_result != 0 )
{
    fprintf(stdout, "getaddrinfo failed with error: %d\n", i_result);
    goto exit;
}

for (ptr=result; ptr != NULL ;ptr=ptr->ai_next)
{
    sock = ECHOCLN_connect(ptr);
    if (sock == RTCS_SOCKET_ERROR)
    {
        continue;
    }
    break;
}
freeaddrinfo(result);

if (sock == RTCS_SOCKET_ERROR)
{
    fprintf(stdout, "Unable to connect to server!\n");
    goto exit;
}
```

7.25 ECHOCLN_process

向服务器发送回显（echo）数据，并接收送回的数据。

概要

```
int32_t ECHOCLN_process(uint32_t sock,
        char   *buffer,
        uint32_t   buflen,
        int32_t      count,
        TIME_STRUCT_PTR  time_ptr)
```

参数

- sock [in] – 连接的套接字句柄
- buffer [in] – 指针，指向要发送的数据
- buflen[in] – 发送数据的字节长度
- count [in] – echo 循环的次数。如果为负值或零，应采用 RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT。
- time_ptr [in/out]输入为非零值表明该函数将测量 echo 循环的持续时间，循环次数由 count 参数指定，得出的时间结果将被写入 time_ptr 指向的 TIME_STRUCT。如果给定值为 NULL，则不测量时间。

说明

一个 echo 循环包括：将给定的数据缓冲区送到给定的套接字。然后，从套接字接收输入数据，并将接收的数据与发送的数据进行比较。循环重复 count 次。返回值

返回值

- RTCSOK（成功）。
- ECHOCLN_ERR_SOCKET（send()或 recv()错误）
- ECHOCLN_ERR_DATA_COMPARE_FAIL（接收的数据与发送的数据不同）
- ECHOCLN_ERR_OUT_OF_MEMORY （无法为接收的数据分配存储器）
- ECHOCLN_ERR_INVALID_PARAM （输入参数无效）

另请参见

- ECHOCLN_connect
- RTCSCFG_ECHOCLN_DEFAULT_BUFLEN
- RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT
- RTCSCFG_ECHOCLN_DEBUG_MESSAGES

示例

```
int32_t i_result;
TIME_STRUCT diff_time;

i_result = ECHOCLN_process(sock, buffer, buflen, loop_cnt, diff_time);
if (RTCS_OK != i_result)
{
   fprintf(stdout, "ECHO client error %i after run time %i.%i\n", i_result,
diff_time.SECONDS, diff_time.MILLISECONDS);
}
else
{
    fprintf(stdout, "exchanged %d echo packets, total time: %i.%i s\n", loop_cnt,
diff_time.SECONDS, diff_time.MILLISECONDS);
}
closesocket(sock);
```

7.26 ECHOSRV_init()

启动 RFC 862 Echo 服务器。该函数接收数据，然后发回给发送者。在服务初始化过程中，针对每个启用的 IP 系列创建一个监听流套接字和一个数据报套接字。支持所有可能组合，如 IPv4、IPv6 以及 IPv4+IPv6。

概要

```
void * ECHOSRV_init(ECHOSRV_PARAM_STRUCT * params)
```

参数

ECHOSRV_PARAM_STRUCT * *params* — 指向 ECHOSRV_PARAM_STRUCT 的指针。

说明

ECHOSRV 通过一般 RTCS 编译/运行时选项、ECHOSRV 特定的编译时选项和 ECHOSRV_PARAM_STRUCT 配置结构进行完全配置。在调用该函数之前，创建并填充 ECHOSRV_PARAM_STRUCT，该结构可存在于应用的任务栈中，也可作为全局变量。当 ECHOSRV_init()函数返回一个非零值后，ECHOSRV_PARAM_STRUCT 结构即被弃用。

返回值

若提供了一个无效的输入参数，该函数返回一个零值 (0)。此情况下会适当地设置任务错误代码。否则，将返回一个有效的非零指针，指向一个新的 ECHOSRV 实例。应用层代码应保存该值，以备用于后续的 ECHOSRV_release()函数中。

另请参见

- ECHOSRV_release()
- ECHOSRV_PARAM_STRUCT
- ECHOSRV 编译时选项 (2.16)

示例

```
#include "echosrv.h"
void * echosrv_ptr;
ECHOSRV_PARAM_STRUCT params = {
     AF_INET | AF_INET6, /* for IPv4+IPv6 */
     7, /* service runs on port 7 by default */
   #if RTCSCFG_ENABLE_IP4
     INADDR_ANY, /* Listening IPv4 address */
   #endif
   #if RTCSCFG_ENABLE_IP6
     IN6ADDR_ANY_INIT, /* Listening IPv6 address */
     0, /* Scope ID for IPv6. 0 is for any Interface. */
   #endif
    7 /* priority of ECHOSRV task */
   };

echosrv_ptr = ECHOSRV_init(&params);
```

7.27 ECHOSRV_release()

停止 RFC 862 Echo 服务器。

概要

uint32_t ECHOSRV_release(void * *server_h*)

参数

void * *server_h* - 指向一个 ECHOSRV 实例的指针。它是 ECHOSRV_init()的返回值。

说明

该函数可关闭所有监听套接字，关闭所有客户端套接字，释放所有内存资源并销毁 ECHOSRV_task。

返回值

如果成功，该函数返回 RTCS_OK。如果

由 *server_h* 输入参数指定的任务在系统中不存在，返回值则等于 RTCSERR_SERVER_NOT_RUNNING。

另请参见

ECHOSRV_init()

ECHOSRV_PARAM_STRUCT

ECHOSRV 编译时选项（2.16）

示例

```
#include "echosrv.h"
uint32_t retval;
retval = ECHOSRV_release(echosrv_ptr);
```

7.28 ENET_get_stats()

获取 RTCS 为以太网接口收集的指向以太网统计数据的指针。

概要

```
    ENET_STATS_PTR ENET_get_stats(
             _enet_handle * handle)
```

参数

handle [in] — 指向以太网句柄的指针

说明

该函数不是 RTCS 的组成部分。如果您正在使用 MQX RTOS, 该函数可供您直接使用。如果您正在将 RTCS 移植到另一个操作系统，则应用必须提供此函数。

返回值

指向 *ENET_STATS* 结构的指针。

另请参见

- ICMP_STATS
- inet_pton()
- RTCS_if_add()
- ENET_STATS

示例

```
ENET_STATS_PTR  enet;
_enet_handle    ehandle;
...
enet = ENET_get_stats();
printf("\n%d Ethernet packets received", enet->ST_RX_TOTAL);
```

7.29 ENET_initialize()

初始化以太网设备接口。

概要

```
    uint32_t ENET_initialize(
            uint32_t device_num,
            _enet_address address,
            uint32_t flags,
            _enet_handle * enet_handle)
```

参数

device_num [in] — 待初始化设备的设备编号。

address [in] — 待初始化设备的以太网地址。

[in] — 以下之一:

非零值（使用设备 EEPROM 中的以太网地址)。

零（使用 *address*)。

此参数不再使用，将被忽略:

enet_handle [out] — 指针，指向设备接口的以太网句柄。

说明

该函数不是 RTCS 的组成部分。如果您正在使用 MQX RTOS, 该函数可供您直接使用。如果您正在将 RTCS 移植到另一个操作系统，则应用必须提供此函数。

注

此函数针对每个设备编号只能调用一次。

此函数执行下列任务:

- 初始化以太网硬件，使其准备好发送和接收以太网数据包。
- 安装以太网中断服务例程。
- 设定发送与接收缓冲区，通常表示以太网设备自身的缓冲区。
- 分配和初始化以太网句柄，供上层从以太网驱动程序 API 和 RTCS API 中调用函数时使用。

返回值

- ENET_OK（成功）
- 以太网错误代码（失败）

示例

请参见示例：设置 RTCS 。

7.30 FTP_close()

终止一个 FTP 会话。

概要

```
int32_t FTP_close(
        pointer     handle,
        FILE_PTR    ctrl_fd)
```

参数

handle [in] — FTP 会话句柄。

ctrl_fd [in] — 要写入控制连接响应的设备。

说明

函数 FTP_close()向 FTP 服务器发出一个 Quit 命令，关闭控制连接，然后释放分配给 FTP 会话句柄的任何资源。

返回值

- FTP 响应代码（成功）
- -1（失败）

示例

向 FTP 服务器发出一个命令。

概要

```
int32_t FTP_command(
        void      *handle,
        char     *command,
        FILE_PTR     ctrl_fd)
```

参数

handle [in] — FTP 会话句柄。

command [in] — FTP 命令。

ctrl_fd [in] — 要写入控制连接响应的设备。

说明

函数 FTP_command()向 FTP 服务器发出一个命令。

返回值

- FTP 响应代码（成功）
- -1（失败）

7.31 FTP_command_data()

向需要数据连接的 FTP 服务器发出一个命令。

概要

```
int32_t FTP_command(
        void     *handle,
        char *command,
        FILE_PTR   ctrl_fd,
        FILE_PTR   data_fd,
        uint32_t    flags)
```

参数

handle [in] — FTP 会话句柄。

command [in] — FTP 命令。

ctrl_fd [in] — 要写入控制连接响应的设备。

data_fd [in] — 用于数据连接的设备。

flags [in] — 用于数据连接的选项。

说明

函数 FTP_command_data()向 FTP 服务器发送一个命令，打开一个数据连接，然后进行数据传输。

参数标志是按位数据或以下项之一:

- 连接模式，必须为下列一种:
  - FTPMODE_DEFAULT — 客户端将默认端口用于数据连接。
  - FTPMODE_PORT — 客户端将选择一个未使用端口并发出一个 PORT 命令。
  - FTPMODE_PASV — 客户端将发出一个 PASV 命令。
- 数据传输方向，必须为下列一种:
  - FTPDIR_RECV — 客户端将从数据连接读取数据，并将其写入 data_fd。
  - FTPDIR_SEND — 客户端将从 data_fd 读取数据，并将其发送到数据连接。

返回值

- FTP 响应代码（成功）
- -1（失败）

7.32 FTP_open()

启动一个 FTP 会话。

概要

```
int32_t FTP_open(
         void *   *handle_ptr,
        _ip_address       server_addr,
        FILE_PTR          ctrl_fd)
```

参数

handle_ptr [in] — FTP 会话句柄。

server_addr [in] — FTP 服务器的 IP 地址。

ctrl_fd [in] — 要写入控制连接响应的设备。

说明

该函数向特定的 FTP 服务器建立一个连接。如果成功，可调用函数 FTP_command() 和 FTP_command_data()向 FTP 服务器发出命令。

返回值

- FTP 响应代码（成功）
- -1 （失败）

示例

```
    #include <mqx.h>
    #include <bsp.h>
    #include <rtcs.h>

    void main_task
       (
          uint32_t dummy
       )
    { /* Body */
      void *ftphandle;
      int32_t response;

      response = FTP_open(&ftphandle, SERVER_ADDRESS, stdout);
      if (response == -1) {
          printf("Couldn't open FTP session\n");
          return;
      } /* Endif */

      response = FTP_command(ftphandle, "USER anonymous\r\n",
         stdout);

      /* response 3xx means Password Required */
      if ((response >= 300) && (response > 400)) {
           response = FTP_command(ftphandle, "PASS password\r\n",
              stdout);
      } /* Endif */

      /* response 2xx means Logged In */
      if ((response >= 200) && (response < 300)) {
           response = FTP_command_data(ftphandle, "LIST\r\n", stdout,
              stdout, FTPMODE_PORT | FTPDIR_RECV);
      } /* Endif */

      FTP_close(ftphandle, stdout);

} /* Endbody */
```

7.33 FTPSRV_init()

启动 FTP 服务器。

概要

```
uint32_t FTPSRV_init(
        FTPSRV_PARAM_STRUCT *params)
```

参数

params[in] — FTP 服务器的参数。

说明

函数 FTPSRV_init()根据_params_结构中的参数启动 FTP 服务器。至少要在该结构中设置一个根目录。如果服务器为非匿名 (默认为非匿名)，则必须设置验证表；否则您无法使用特权服务器命令。关于每个服务器参数的更多说明，请参见“FTPSRV_PARAM_STRUCT”章节。

返回值

- 非零值 （成功）
- 零 （失败）

示例

```
#include "ftpsrv.h"
static const FTPSRV_AUTH_STRUCT ftpsrv_users[] =
    {
        {"developer", "freescale", NULL},
        {NULL,      NULL,        NULL}
    };
FTPSRV_PARAM_STRUCT params;
    uint32_t handle;
    _mem_zero(&params, sizeof(params));
    params.auth_table = (FTPSRV_AUTH_STRUCT*) ftpsrv_users;
    params.root_dir = "a:";
    handle = FTPSRV_init(params);
```

另请参见

- FTPSRV_release
- FTPSRV_PARAM_STRUCT

7.34 FTPSRV_release

停止 FTP 服务器并释放其所有资源。

概要

```
uint32_t FTPSRV_init(
          FTPSRV_PARAM_STRUCT *params)
```

参数

params[in] — FTP 服务器的参数。

说明

该函数的操作与 FTPSRV_init()相反。它关闭所有正在监听的套接字，停止所有服务器任务并释放服务器使用的所有内存。将阻塞调用任务，直到服务器停止且资源释放为止。

返回值

- RTCS_OK - 成功关闭。
- RTCS_ERR - 关闭失败。

另请参见

- FTPSRV_init()

7.35 getaddrinfo()

获取人类可读的主机名或地址的 IP 地址列表。

概要

```
int32_t getaddrinfo(const char *hostname, const char *servname, const struct addrinfo
*hints, struct addrinfo **res)
```

参数

hostname [in] — 要解析的主机名。这可能是一个主机名或数字形式的主机地址串（点分十进制 IPv4 地址或十六进制 IPv6 地址）。

servname [in] — 端口号字符串。

hints [in] — 指针，指向的 addrinfo 结构可用于提示套接字的类型。为可选参数(0)。

res [out] — 地址，表示函数用来存放指针的位置，该指针指向 addrinfo 结构的结果链表。

返回值

如果成功，返回零；出现错误则返回非零。

说明

该函数用于获取主机名 hostname 或服务 servname 的 IP 地址和端口号列表。

hostname 和 servname 参数为指向以零结尾的字符串的指针或零指针。hostname 可以是一个有效的主机名或数字形式的主机地址串(包含一个点分十进制 IPv4 地址或一个 IPv6 地址)。servname 为十进制端口号。hostname 和 servname 中必须至少有一个非零。

hints 是一个指向结构 addrinfo 的指针，可选。

```
struct addrinfo {
    uint16_t        ai_flags;          /* input flags */
    uint16_t        ai_family;         /* protocol family for socket */
    uint32_t        ai_socktype;       /* socket type */
    uint16_t        ai_protocol;       /* protocol for socket */
    unsigned int    ai_addrlen;        /* length of socket-address */
    char            *ai_canonname;      /* canonical name for service location */
    struct sockaddr  *ai_addr;           /* socket-address for socket */
    struct addrinfo  *ai_next;           /* pointer to next in list */
};
```

该结构可用于提供调用方支持的或希望使用的套接字类型信息。调用方在提示信息中可提供以下结构元素:

- ai_family - 应使用的协议系列 (AF_INET、AF_INET6、AF_UNSPEC)。当 ai_family 设置为 AF_UNSPEC 时，意味着调用方将接受 TCP/IP 协议栈支持的任何协议系列。
- ai_socktype - 表示所需的套接字类型：SOCK_STREAM 或 SOCK_DGRAM。若 ai_socktype 为零，调用方将会接受任何套接字类型。
- ai_protocol - 表示需要何种传输协议：IPPROTO_UDP 或 IPPROTO_TCP。若 ai_protocol 为零，调用方将会接受任何协议。
- ai_flags - hints 参数指向的 ai_flags 字段应设为零或按位包含，或值 AI_CANONNAME、AI_NUMERICHOST 和 AI_PASSIVE 中的一个或多个:
  - AI_CANONNAME - 如果 AI_CANONNAME 位置位，成功调用 getaddrinfo() 将返回一个以零结尾的字符串，包含 addrinfo 结构返回的 ai_canonname 元素中指定 hostname 的规范名称。
  - AI_NUMERICHOST - 如果 AI_NUMERICHOST 位置位，说明 hostname 应作为定义 IPv4 或 IPv6 地址的数字字符串处理，不应尝试名称解析。
  - AI_PASSIVE - 如果 AI_PASSIVE 位置位，说明返回的套接字地址结构用于调用 bind(2)。这种情况下，如果 hostname 参数为零指针，则套接字地址结构的 IP 地址部分将设为 INADDR_ANY 以用于 IPv4 地址，或设为 IN6ADDR_ANY_INIT 以用于 IPv6 地址。如果 AI_PASSIVE 位未置位，对于面向连接的协议，返回的套接字地址结构将准备用于调用 connect()，若选择了无连接协议，将准备用于调用 connect()、sendto()或 sendmsg()。如果 hostname 为空指针且 AI_PASSIVE 未置位，则套接字地址结构的 IP 地址部分将设为环回地址。

通过 hints 传递的所有其他 addrinfo 结构元素必须为零或空指针。

如果 hints 为空指针，getaddrinfo()的表现就像调用方提供的 addrinfo 结构使 ai_family 设为 AF_UNSPEC，其他元素均设为零。

成功调用 getaddrinfo()后，*res 指针指向含有一个或多个 addrinfo 结构的链接列表。可通过跟随每个 addrinfo 结构中的 ai_next 指针遍历该列表，直到遇到空指针。每个返回的 addrinfo 结构中的 ai_family、ai_socktype 和 ai_protocol 成员适用于调用 socket()。对于列表中的每个 addrinfo 结构，ai_addr 成员指向一个长度为 ai_addrlen 的已填充套接字地址结构。

getaddrinfo()的这种实现方式允许 IPv6 数字地址表示法带有范围标识符，其形式为 <address>%<zone-id>。通过向地址添加百分符和范围标识符，用户可以填写地址的 sin6_scope_id 字段。

getaddrinfo()返回的所有信息均动态分配：addrinfo 结构本身以及 addrinfo 结构中包含的套接字地址结构和规范的主机名字符串。

成功调用 getaddrinfo()创建的动态分配结构所分配的内存由 freeaddrinfo()函数释放。

示例

```
      {
      struct addrinfo    *addrinfo_result;
      struct addrinfo    *addrinfo_result_first;
      int32_t            retval;
      char               addr_str[RTCS_IP6_ADDR_STR_SIZE];

      _mem_zero(&addrinfo_hints, sizeof(addrinfo_hints));
      addrinfo_hints.ai_flags = AI_CANONNAME;

      retval = getaddrinfo("www.example.com", NULL, NULL, &addrinfo_result);
      if (retval == 0)
      {

          addrinfo_result_first = addrinfo_result;
          /* Print all resolved IP addresses.*/
          while(addrinfo_result)
          {
              if(inet_ntop(addrinfo_result->ai_family,
                           &((struct sockaddr_in6 *)((*addrinfo_result).
ai_addr))->sin6_addr,
                           addr_str, sizeof(addr_str)))
              {
                 printf("\t%s\n", addr_str);
              }
              addrinfo_result = addrinfo_result->ai_next;
        }

        freeaddrinfo(addrinfo_result_first);
    }
    else
    {
        printf("Unable to resolve host\n");
    }
 }
```

7.36 freeaddrinfo()

释放 getaddrinfo()分配的存储器。

概要

```
void freeaddrinfo(struct addrinfo *ai);
```

参数

ai [in] — 指向 addrinfo 结构链表的指针。

返回值

- 无。

说明

该函数释放由 getaddrinfo()分配的 addrinfo 结构，包括带有指向（ai_canonname 和 ai_addr）的 addrinfo 结构成员的缓冲区 。

示例

```
{
     struct addrinfo    *addrinfo_result;
     struct addrinfo    *addrinfo_result_first;
     int32_t            retval;
     char               addr_str[RTCS_IP6_ADDR_STR_SIZE];

    _mem_zero(&addrinfo_hints, sizeof(addrinfo_hints));
    addrinfo_hints.ai_flags = AI_CANONNAME;

    retval = getaddrinfo("www.example.com", NULL, NULL, &addrinfo_result);
    if (retval == 0)
    {

       addrinfo_result_first = addrinfo_result;
       /* Print all resolved IP addresses.*/
       while(addrinfo_result)
       {
           if(inet_ntop(addrinfo_result->ai_family,
                 &((struct sockaddr_in6 *)((*addrinfo_result).ai_addr))->sin6_addr,
                      addr_str, sizeof(addr_str)))
           {
              printf("\t%s\n", addr_str);
           }
           addrinfo_result = addrinfo_result->ai_next;
       }

       freeaddrinfo(addrinfo_result_first);
    }
    else
    {

        printf("Unable to resolve host\n");
    }
 }
```

7.37 getnameinfo()

提供从地址到名称的名称解析。

概要

```
int32_t getnameinfo( const struct sockaddr *sa, unsigned int salen, char *host, unsigned int
hostlen, char *serv, unsigned int servlen, int flags)
```

参数

sa [in] — 指针，指向要转换的套接字地址结构。其中含有地址和端口号。

salen [in] — sa 所指的套接字地址结构的字节长度。

host [out] — 指针，指向一个字符串缓冲区以存放返回的主机名。可选（零)。

hostlen [in] — host 所指的字符串缓冲区的字节长度，包括零终止符。

serv [out] — 指针，指向一个字符串缓冲区以存放返回的端口号。可选（零)。

servlen [in] — serv 所指的字符串缓冲区的字节长度，包括零终止符。

flags [in] — 标志参数，用来修改 getnameinfo()函数的行为。

返回值

- 如果成功，返回零；出现错误则返回非零。

说明

该函数用于将一个套接字地址转换为一个主机名和端口号。

host 参数指向一个可容纳多达 hostlen 个字符的缓冲区，如果 host 参数为非零，hostlen 参数为非零，则缓冲区将主机名作为一个以零结尾的字符串接收。如果 host 参数或 hostlen 参数为零，则不应返回主机名。如果无法找到主机的名称，则不返回名称，而返回 sa 参数所指的套接字地址结构中所含的数字地址。

service 参数指向一个可容纳多达 servlen 个字节的缓冲区，如果 serv 参数为非零，servlen 参数为非零，则缓冲区将端口号作为一个以零结尾的字符串接收。如果 serv 参数或 servlen 参数为零， 则不返回端口号字符串。

这些标志参数用来修改函数的行为:

- NI_NOFQDN — 如果置位，仅返回 FQDN（完全限定的域名）的 hostname 部分。

- NI_NUMERICHOST — 如果置位，则返回 hostname 的数字形式。若没有置位，在节点名称无法确定的情况下仍会出现此行为。该函数允许 IPv6 数字地址表示法带有范围标识符。
- NI_NAMEREQD — 如果置位，若 hostname 无法确定，将返回一个错误。

getnameinfo()函数找到 getaddrinfo()的反函数，并替换已废弃 gethostbyaddr()的功能。

示例

```
{
    struct addrinfo     *addrinfo_result;
    struct addrinfo     *addrinfo_result_first;
    int32_t             retval;
    char                addr_str[NI_MAXHOST];

    _mem_zero(&addrinfo_hints, sizeof(addrinfo_hints));
    addrinfo_hints.ai_flags = AI_CANONNAME;

    retval = getaddrinfo("www.example.com", NULL, NULL, &addrinfo_result);
    if (retval == 0)
    {
       addrinfo_result_first = addrinfo_result;
       /* Print all resolved IP addresses.*/
       while(addrinfo_result)
       {
            /* Print numeric form of the address.*/
         if(getnameinfo(addrinfo_result->ai_addr,
                        addrinfo_result->ai_addrlen,
                            host_str, sizeof(host_str),


                            NULL, 0, NI_NUMERICHOST) == 0)
         {
             printf("\t%s\n", addr_str);
         }
         addrinfo_result = addrinfo_result->ai_next;
      }

      freeaddrinfo(addrinfo_result_first);
  }
  else
  {
     printf("Unable to resolve host\n");
  }
}
```

7.38 getpeername()

获取套接字的远程端点标识符。

概要

```
uint32_t    getpeername(
         uint32_t          socket,
         sockaddr     *    name,
         uint16_t     *    namelen)
```

参数

socket [in] — 用于流套接字的句柄。

name [out] — 指针，指向用于套接字远程端点标识符的占位符。

namelen [in/out] — 当传入时：为指针，指向 name 所指内容的字节长度。

当传出时：为远程端点标识符的完整大小，以字节为单位。

说明

函数 getpeername()可找到由 connect()或 accept()确定的套接字的远程端点标识符。该函数阻塞，但是会立即执行并回复命令。

返回值

- RTCS_OK（成功）
- 特定错误代码（失败）

示例

```
uint32_t    handle;
sockaddr_in remote_sin;
uint32_t    status;
uint16_t    namelen;

...

namelen = sizeof (sockaddr_in);
status = getpeername(handle, (struct sockaddr *)&remote_sin, &namelen);
if (status != RTCS_OK)
{
  printf("\nError, getpeername() failed with error code %lx",
     status);
} else {
  printf("\nRemote address family is %x", remote_sin.sin_family);
  printf("\nRemote port is %d", remote_sin.sin_port);
  printf("\nRemote IP address is %lx",
     remote_sin.sin_addr.s_addr);
}
```

7.39 getsockname()

获取套接字的本地端点标识符

概要

```
uint32_t   getsockname(
          uint32_t         socket,
          sockaddr     *   name,
          uint16_t     *   namelen)
```

参数

socket [in] — 套接字句柄。

name [out] — 指针，指向用于套接字远程端点标识符的占位符。

namelen [in/out] — 当传入时：为指针，指向 name 所指内容的字节长度。

当传出时：为远程端点标识符的完整大小，以字节为单位。

说明

函数 getsockname()返回由 bind()定义的套接字的本地端点。该函数阻塞，但是会立即执行并回复命令。

返回值

- RTCS_OK（成功）
- 特定错误代码（失败）

示例

```
uint32_t                                        handle;
sockaddr_in   local_sin;
uint32_t       status;
uint16_t       namelen;

...
namelen = sizeof (sockaddr_in);
status = getsockname(handle, (struct sockaddr *)&local_sin, &namelen);

if (status != RTCS_OK)
{
  printf("\nError, getsockname() failed with error code %lx",
         status);
} else {
  printf("\nLocal address family is %x", local_sin.sin_family);
  printf("\nLocal port is %d", local_sin.sin_port);
  printf("\nLocal IP address is %lx", local_sin.sin_addr.s_addr);
}
```

7.40 getsockopt()

获取套接字选项的值。

概要

```
uint32_t   getsockopt(
         uint32_t          socket,
         int32_t           level,
         uint32_t          optname,
         void              *optval,
         uint32_t    *    o ptlen)
```

参数

socket [in] — 套接字句柄。

level [in] — 选项所在的协议层。

optname [in] — 选项名称 (见说明)。

optval [in/out] — 指向选项值的指针。

optlen [in/out] — 当传入时：为 optval 的字节长度。

当传出时：为选项值的整体字节长度。

说明

一个应用可获取所有协议层的所有套接字选项。关于套接字选项和协议层的详细说明，请参见 setsockopt()。该函数阻塞，但是会立即执行并回复命令。

返回值

- RTCS_OK (成功)
- 特定错误代码 (失败)

7.41 HTTPSRV_init()

该函数可初始化并启动 HTTP 服务器。

概要

```
uint32_t HTTPSRV_init(
HTTPSRV_PARAM_STRUCT *params);
```

参数

params [in] — 指针，指向由 HTTP 函数使用的参数结构。可为零 — 在此情况下采用默认值。设为零的参数将被忽略，并且采用默认值。

说明

这是用于初始化并启动服务器的主要 HTTP 函数。通过参数提供的信息来分配内部存储器缓冲区，设置套接字和会话。

不得在运行时更改任何以指针形式传递给服务器的参数，因为这样可能会导致内存破坏和其他意外后果。若要更改服务器的设置，则必须先使用函数 HTTPSRV_release()停止服务器，然后再采用新参数将其启动。

返回值

- 如果成功，返回值为 HTTP 服务器句柄；否则，返回值为零。

另请参见

- HTTPSRV_PARAM_STRUCT

示例

```
#include "httpsrv.h"

HTTPSRV_PARAM_STRUCT params;

_mem_zero(&params, sizeof(params));
params.root_dir = "tfs:";
params.index_page = "\\index.html";
server = HTTPSRV_init(&params);
…
HTTPSRV_release(server);
```

7.42 HTTPSRV_release()

该函数可停止服务器并释放所有分配给服务器的资源。

概要

```
uint32_t HTTPSRV_release(
uint32_t server_h);
```

参数

server_h [in] — 服务器句柄，由 HTTPSRV_init()创建。

说明

当用户应用需要停止服务器时，应该调用此函数。其作用与 HTTPSRV_init()相反。它关闭所有正在监听的套接字，停止所有服务器任务并释放服务器使用的所有存储器。该函数在完成关闭之前保持阻塞状态。

返回值

- 如果关闭成功，返回值为 HTTPSRV_OK ，否则为 HTTPSRV_ERR 。

7.43 HTTPSRV_cgi_write()

该函数用于从 CGI 回调函数向客户端写入数据。

概要

```
uint32_t httpsrv_cgi_write(
HTTPSRV_CGI_RES_STRUCT* response)
```

参数

response [in] — 载有数据的 CGI 响应。该结构中的所有变量都必须设定。

说明

如果用户想要从 CGI 回调函数内部向客户端发送一个响应，则需使用此函数。须在调用 HTTPSRV_cgi_write()之前创建并设定响应结构。首次调用后，HTTP 服务器根据响应中的值生成一个标头，然后将其保存到会话缓冲区或发送到客户端，取决于缓冲区的状态。响应中的任何数据都会被处理（发送/储存）。之后每次的调用仅写入响应结构中数据变量所指的数据。

请注意，如果您启用了持久连接功能，并将响应结构中的 content_length 变量设定为零，对于活动会话，持久连接功能将自动禁用。相关原因请参见 RFC2616 的 4.4 节（http://tools.ietf.org/html/rfc2616#section-4.4）。

返回值

服务器成功处理的字节数。

另请参见

- HTTPSRV_CGI_RES_STRUCT

示例

关于如何使用该函数的详细示例，请参见文件%MQX_PATH%\rtcs\examples\httpsrv\cgi.c （可复制该链接并粘贴至浏览器的地址栏）。

7.44 HTTPSRV_cgi_read()

该函数用于读取由客户端提供、作为 CGI 回调函数中实体的数据。

概要

```
uint32_t httpsrv_cgi_read(
uint32_t ses_handle,
char* buffer,
uint32_t length);
```

参数

ses_handle [in] — 从 CGI 请求结构体得到的会话句柄。

buffer [in] — 指针，指向从服务器读取数据的缓冲区。

length [in] — 缓冲区的字节长度。

说明

在用户 CGI 脚本需要从客户端读取数据时将调用该函数。

返回值

读取的字节数。

示例

关于如何使用该函数的详细示例，请参见文件 *%MQX_PATH%\demo\web_hvac\cgi_hvac.c* (可复制该链接并粘贴至浏览器的地址栏) 。该函数应具有一个“length”参数的返回值。如果其返回值较小，表示从套接字读取数据时出错，则不应在相同上下文中使用相同的句柄再次调用该函数。

7.45 HTTPSRV_ssi_write()

该函数用于从服务器端包含函数向客户端写入数据。

概要

```
HTTPSRV_ssi_write(
uint32_t ses_handle,
char* data,
uint32_t length)
```

参数

ses_handle [in] — 会话句柄。该句柄是从 SSI 参数结构复制来的值。

data [in] — 指针，指向要发送到客户端的数据。

length [in] — 数据的字节长度。

说明

所有传递到该函数的数据都作为 HTTP 的响应内容发送到客户端。

返回值

写入的字节数。

示例

```
#include "httpsrv.h"
static _mqx_int usb_status_fn(HTTPSRV_SSI_PARAM_STRUCT* param)
{
     char* str;

     if (usbstick_attached())
     {
        str = "visible";
     }
     else
     {
        str = "hidden";
     }
     HTTPSRV_ssi_write(param->ses_handle, str, strlen(str));
     return 0;
}
```

7.46 LLMNRSRV_init

启动本地链路多播名称解析（LLMNR）服务器。

概要

```
uint32_t LLMNRSRV_init(LLMNRSRV_PARAM_STRUCT *params)
```

参数

- *params[in]* – 初始化 LLMNR 服务器参数。

说明

该函数根据初始化参数结构启动 LLMNR 服务器。在该结构中，网络接口和主机名表为必填参数，所有其他为可选参数并可设为零。关于每个服务器参数的说明，请参见 LLMNRSRV_PARAM_STRUCT。

返回值

- 如果初始化成功，服务器开始处理，否则返回零。

示例

```
LLMNRSRV_PARAM_STRUCT     llmnr_params;
LLMNRSRV_HOST_NAME_STRUCT host_name_table[] ={ {"rtcs",0},
                                               {0,0} /*end mark*/}
_rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);

_mem_zero(&llmnr_params, sizeof(llmnr_params));
llmnr_params.interface = ihandle;
llmnr_params.host_name_table = host_name_table;

/* Start LLMNR server.*/
llmnr_desc = LLMNRSRV_init(&llmnr_params;
```

另请参见

- LLMNRSRV_release
- LLMNRSRV_PARAM_STRUCT

7.47 LLMNRSRV_release

停止 LLMNR 服务器并释放其所有资源。

概要

```
uint32_t LLMNRSRV_release(uint32_t server_h)
```

参数

- *server_h[in]* - 服务器句柄（由 LLMNRSRV_init 返回）。

说明

该函数的操作与 LLMNRSRV_init()相反。它用于停止服务器任务，并释放由服务器分配的所有资源。将阻塞调用任务，直到服务器停止且资源被释放为止。

返回值

- RTCS_OK

另请参见

- LLMNRSRV_init

7.48 ICMP_stats()

获取指向 ICMP 统计数据的指针。

概要

```
ICMP_STATS_PTR ICMP_stats(void)
```

说明

函数 ICMP_stats()无参数，返回一个指向 ICMP 统计数据的指针，该统计数据由 RTCS

收集。

返回值

指向 ICMP_STATS 结构的指针。

另请参见

- TCP_stats()
- ICMP_STATS

7.49 IGMP_stats()

获取指向 IGMP 统计数据的指针。

概要

```
IGMP_STATS_PTR IGMP_stats(void)
```

说明

函数 IGMP_stats()无参数，返回一个指向 IGMP 统计数据的指针，该统计数据由 RTCS 收集。

返回值

指向 IGMP_STATS 结构的指针。

另请参见

- TCP_stats()
- IGMP_STATS

7.50 inet_pton()

该函数将字符串 src 转换成一个网络地址结构。

概要

```
uint32_t inet_pton (
          int32_t af,
          const char *src,
          void *dst,
          unsigned int sizeof_dst)
```

参数

af [in] — 系列名。

**src[in]* — 指针，指向 prn 格式地址。

**dst[out]* — 指针，指向 bin 格式的地址。

sizeof_dst [in] — dst 缓冲区的大小。

说明

该函数将字符串 src 转换成 af 地址系列中的一个网络地址结构，然后将网络地址结构复制到 dst。af 参数必须为 AF_INET 或 AF_INET6。目前支持以下地址系列:

AF_INET

src 指向一个含有 IPv4 网络地址的字符串，点分十进制格式“ddd.ddd.ddd.ddd”，其中 ddd 为范围从 0 到 255 的三位十进制数。该地址转换为结构 in_addr，并被复制到 dst，其所占大小（结构 in_addr）必须为 4 个字节（32 位）。

AF_INET6

src 指向一个含有 IPv6 网络地址的字符串。该地址转换为结构 in6_addr 并被复制到 dst，其所占大小（结构 in6_addr）必须为 16 个字节（128 位）。IPv6 地址允许的格式符合下列规则：

格式为 x:x:x:x:x:x:x:x。该格式中包含 8 个十六进制数，每个数字分别代表一个 16 位值（即每个 x 可表示 4 个 hex 数字）。在首选格式中，一系列连续的零值可简略为::。一个地址中只能出现一个::实例。例如，环回地址 0:0:0:0:0:0:0:1 可简略为::1。通配符地址全零，可写为::。

返回值

- RTCS_OK（成功）
- RTCS_ERROR（失败）

示例

IPv4 协议。

```
uint32_t temp;
inet_pton (AF_INET, prn_addr, &temp, sizeof(temp));
```

IPv6 协议。

```
in6_addr addr6;
inet_pton (AF_INET6,"abcd:ef12:3456:789a:bcde:f012:192.168.24.252",&addr6);
```

7.51 inet_ntop()

将地址*src 从网络格式（通常为结构 in_addr 或 in6addr，按照网络字节顺序排列）转换为适合外部显示用途的显示格式。

概要

```
char *inet_ntop(
         int32_t af,
         const void *src,
         char *dst,
         socklen_t size)
```

参数

af [in] — 系列名。

**src[in]* — 指针，指向网络格式的地址。

**dst[out]* — 指针，指向显示格式的地址。

sizeof_dst [in] — dst 缓冲区的大小。

说明

将地址*src 从网络格式（通常为结构 in_addr 或 in6addr，按照网络字节顺序排列）转换为显示格式（适合外部显示用途）。该函数目前对于

AF_INET 和 AF_INET6 有效。

返回值

如果系统出现错误，该函数返回一个零值，否则该函数返回一个指向目标字符串的指针。

示例

IPv4 协议。

```
in_addr addr;
char prn_addr[RTCS_IP4_ADDR_STR_SIZE];
.......
inet_ntop(AF_INET, &addr, prn_addr, sizeof(prn_addr));
printf("IP addr = %s\n", prn_addr);
.......
```

IPv6 协议。

```
in6_addr addr6;
char prn_addr6[RTCS_IP6_ADDR_STR_SIZE];
.......
inet_ntop(AF_INET6,&addr6, prn_addr6, sizeof(prn_addr6));
printf("IP addr = %s\n",prn_addr6);
.......
```

7.52 IP_stats()

获取指向 IP 统计数据的指针。

概要

```
IP_STATS_PTR IP_stats(void)
```

说明

函数 IP_stats()无参数，返回一个指向 IP 统计数据的指针，该统计数据由 RTCS 收集。

返回值

指向 IP_STATS 结构的指针。

另请参见

- TCP_stats()
- IP_STATS

7.53 IPIF_stats()

获取 RTCS 为设备接口收集的 IPIF 统计数据的指针。

概要

```
IPIF_STATS_PTR IPIF_stats(
         _rtcs_if_handle rtcs_if_handle)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

说明

函数 IPIF_stats()可获取 RTCS 为设备接口收集的 IPIF 统计数据的指针。

返回值

- 指向 IPIF_STATS 结构的指针（成功）
- 零（失败：rtcs_if_handle 无效）

另请参见

- TCP_stats()
- IPIF_STATS

7.54 ipcfg_init_device()

初始化以太网设备，添加网络接口，并为其建立 IPCFG 上下文。

概要

```
uint32_t ipcfg_init_device(
         uint32_t device,
         _enet_address mac)
```

参数

device [in] — 设备标识（索引）

mac [in] — 以太网 MAC 地址

说明

该函数可初始化以太网设备（内部调用 ENET_initialize），添加 RTCS 的网络接口（RTCS_if_add），并为设备建立 IPCFG 上下文。

返回值

- IPCFG_OK （成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT

另请参见

- RTCS_if_add()

示例

```
#define ENET_IPADDR IPADDR(192,168,1,4)
#define ENET_IPMASK IPADDR(255,255,255,0)
#define ENET_IPGATEWAY IPADDR(192,168,1,1)
uint32_t setup_network(void)
{
   uint32_t              error;
   IPCFG_IP_ADDRESS_DATA ip_data;
   _enet_address         enet_address;

   ip_data.ip = ENET_IPADDR;
   ip_data.mask = ENET_IPMASK;
   ip_data.gateway = ENET_IPGATEWAY;

   /* Create TCP/IP task */
   error = RTCS_create();
   if (error) return error;

   /* Get the Ethernet address of the device */
   ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

   /* Initialize the Ehternet device */
   error = ipcfg_init_device (BSP_DEFAULT_ENET_DEVICE, enet_address);
   if (error) return error;

   /* Bind Ethernet device to network using constant (static) IP address  information */
   error = ipcfg_bind_staticip(BSP_DEFAULT_ENET_DEVICE, &ip_data);
   if (error) return error;

   return 0;
}
```

7.55 ipcfg_init_interface()

为已初始化的设备和其接口建立 IPCFG 上下文。

概要

```
uint32_t ipcfg_init_interface(
         uint32_t device_number,
         _rtcs_if_handle ihandle)
```

参数

device_number [in] — 设备编号

ihandle [in] — 接口句柄

说明

该函数为已被其他 RTCS 调用函数初始化的网络接口建立 IPCFG 上下文。

返回值

- IPCFG_OK （成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT

示例

```
#define ENET_IPADDR IPADDR(192,168,1,4)
#define ENET_IPMASK IPADDR(255,255,255,0)
#define ENET_IPGATEWAY IPADDR(192,168,1,1)
uint32_t setup_network(void)
{
   uint32_t              error;
   IPCFG_IP_ADDRESS_DATA ip_data;
   _enet_address         enet_address;
   _enet_handle          ehandle;
   _rtcs_if_handle       ihandle;

   ip_data.ip = ENET_IPADDR;
   ip_data.mask = ENET_IPMASK;
   ip_data.gateway = ENET_IPGATEWAY;

   error = RTCS_create();
   if (error) return error;

   ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

   error = ENET_initialize(BSP_DEFAULT_ENET_DEVICE, enet_address, 0, &ehandle);
   if (error) return error;

   error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
   if (error) return error;

   error = ipcfg_init_interface(BSP_DEFAULT_ENET_DEVICE, ihandle);
   if (error) return error;

   return ipcfg_bind_autoip(BSP_DEFAULT_ENET_DEVICE, &ip_data);
```

7.56 ipcfg_bind_boot()

使用启动协议将以太网设备绑定到网络。

概要

```
uint32_t ipcfg_bind_boot(
        uint32_t device)
```

参数

device [in] — 设备标识

说明

该函数尝试使用启动协议将以太网设备绑定到网络。它还收集关于 TFTP 服务器和待下载文件的信息。这是一个阻塞函数，也就是直到进程结束或出现错误时才会返回。

绑定过程中出现任何故障都会导致网络接口处于未绑定状态。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

示例

```
#define ENET_IPADDR IPADDR(192,168,1,4)
#define ENET_IPMASK IPADDR(255,255,255,0)
#define ENET_IPGATEWAY IPADDR(192,168,1,1)
uint32_t setup_network(void)
{
    uint32_t               error;
    _enet_address         enet_address;

    error = RTCS_create();
    if (error) return error;

    ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

    error = ipcfg_init_device(BSP_DEFAULT_ENET_DEVICE, enet_address);
    if (error) return error;

    error = ipcfg_bind_boot(BSP_DEFAULT_ENET_DEVICE);
    if (error) return error;

    TFTTIP = ipcfg_get_tftp_serveraddress(BSP_DEFAULT_ENET_DEVICE);
```

```
    TFTPserver = ipcfg_get_tftp_servername(BSP_DEFAULT_ENET_DEVICE);
    TFTPfile = ipcfg_get_boot_filename(BSP_DEFAULT_ENET_DEVICE);
}
```

7.57 ipcfg_bind_dhcp()

将以太网设备绑定到网络，使用 DHCP 协议（轮询模式）。

概要

```
uint32_t ipcfg_bind_dhcp(
         uint32_t device,
         bool try_auto_ip)
```

参数

device [in] — 设备标识

try_auto_ip [in] — 如果 DHCP 绑定失败，尝试使用 auto-ip 自动分配地址。

说明

该函数使用 DHCP 协议发起将设备绑定到网络的过程。由于 DHCP 地址解析可能需要长达一分钟，因此有两个无阻塞函数与 DHCP 绑定相关。

首先必须反复调用 ipcfg_bind_dhcp()，直到其返回的结果不等于 RTCSERR_IPCFG_BUSY。如果其返回 IPCFG_OK，再通过周期性调用 ipcfg_poll_dhcp() 使该过程继续，直到其结果不等于 RTCSERR_IPCFG_BUSY。

这两个函数的前两个参数在调用时都必须使用相同的值。

根据第二个参数，DHCP 绑定失败后可使用附加的自动 IP 绑定方式。

如果这两个过程任何一个返回的结果不等于 RTCS_OK 或 RTCSERR_IPCFG_BUSY，轮询过程应终止。在此情况下，网络接口仍处于未绑定状态。

另一个阻塞型 DHCP 绑定方法是 ipcfg_bind_dhcp_wait() 。关于该函数调用如何在内部实现，请参见如下示例。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER

- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

另请参见

- ipcfg_poll_dhcp()

示例

```
uint32_t ipcfg_bind_dhcp_wait(uint32_t device, bool try_auto_ip,
         IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
{
  uint32_t result = IPCFG_OK;
  do
  {
    if (result == RTCSERR_IPCFG_BUSY) _time_delay(200);
    result = ipcfg_bind_dhcp(device, try_auto_ip);
  } while (result == RTCSERR_IPCFG_BUSY);
  if (result != IPCFG_OK) return result;
  do
  {

   _time_delay (200);
   result = ipcfg_poll_dhcp(device, try_auto_ip, auto_ip_data);
  } while (result == RTCSERR_IPCFG_BUSY);
  return result;
}
```

7.58 ipcfg_bind_dhcp_wait()

将以太网设备绑定到网络，使用 DHCP 协议或阻塞模式。

概要

```
uint32_t ipcfg_bind_dhcp_wait(
         uint32_t device,
         bool try_auto_ip,
         IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
```

参数

device [in] — 设备标识。

try_auto_ip [in] — 如果 DHCP 绑定失败，尝试使用 auto-ip 自动分配地址。

auto_ip_data [in] — 用于 auto-IP 绑定的 IP、掩码和网关信息（可能为 NULL）。

说明

该函数尝试使用 DHCP 协议将设备绑定到网络，或者在 DHCP 绑定失败后使用自动 IP 绑定。这是一个阻塞函数，也就是直到进程结束或出现错误时才会返回。

根据第二个参数, DHCP 绑定失败后可使用附加的自动 IP 绑定方式。若第三个参数为零，则将上一次成功的绑定信息作为自动 IP 绑定的输入。

绑定过程中出现任何故障都会导致网络接口处于未绑定状态。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

另请参见

- ipcfg_bind_dhcp()
- ipcfg_poll_dhcp()

示例

```
#define ENET_IPADDR IPADDR(192,168,1,4)
#define ENET_IPMASK IPADDR(255,255,255,0)
#define ENET_IPGATEWAY IPADDR(192,168,1,1)
uint32_t setup_network(void)
{
    uint32_t error;
    IPCFG_IP_ADDRESS_DATA auto_ip_data;
    _enet_address enet_address;

    auto_ip_data.ip = ENET_IPADDR;
    auto_ip_data.mask = ENET_IPMASK;
    auto_ip_data.gateway = ENET_IPGATEWAY;

    error = RTCS_create();
    if (error) return error;

    ENET_get_mac_address (BSP_DEFAULT_ENET_DEVICE, ENET_IPADDR, enet_address);

    error = ipcfg_init_device(BSP_DEFAULT_ENET_DEVICE, enet_address);
    if (error) return error;

    return ipcfg_bind_dhcp_wait(BSP_DEFAULT_ENET_DEVICE, TRUE, &auto_ip_data);
}
```

7.59 ipcfg_bind_staticip()

将以太网设备绑定到网络，使用常量或静态 IPv4 地址信息。

概要

```
uint32_t ipcfg_bind_staticip(
         uint32_t device,
         IPCFG_IP_ADDRESS_DATA_PTR static_ip_data)
```

参数

device [in] — 设备标识

static_ip_data [in] — 指向 IP、掩码和网关结构的指针

说明

该函数尝试使用给定 IPv4 地址信息将设备绑定到网络。如果地址已使用, 将返回一个错误。这是一个阻塞函数, 也就是直到进程结束或出现错误时才会返回。

绑定过程中出现任何故障都会导致网络接口处于未绑定状态。

返回值

- IPCFG_OK (成功)
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

7.60 ipcfg_get_device_number()

针对给定的 RTCS 接口返回以太网设备编号。

概要

```
uint32_t ipcfg_get_device_number(
         _rtcs_if_handle ihandle)
```

参数

ihandle [in] — 接口句柄

说明

简单函数，通过提供 RTCS 接口句柄返回以太网设备编号。

返回值

如果成功，返回设备编号，否则返回-1。

另请参见

- ipcfg_get_ihandle()

7.61 ipcfg_add_interface()

添加新接口并返回对应的设备编号。

概要

```
uint32_t ipcfg_add_interface(
         uint32_t device_number,
         _rtcs_if_handle ihandle)
```

参数

device_number [in] — 设备编号

ihandle [in] — 接口句柄

说明

该函数建立 ihandle 和设备编号的内部关联。

返回值

如果成功，返回设备编号，否则返回-1。

另请参见

- ipcfg_get_ihandle()

7.62 ipcfg_get_ihandle()

针对给定的以太网设备编号返回 RTCS 接口句柄。

概要

```
_rtcs_if_handle ipcfg_get_ihandle(
     uint32_t device)
```

参数

device [in] — 设备标识

说明

简单函数，通过提供以太网设备编号返回 RTCS 接口句柄。

返回值

如果成功，返回接口句柄，否则返回零值。

7.63 ipcfg_get_mac()

返回以太网 MAC 地址。

概要

```
bool ipcfg_get_mac(
        uint32_t device,
        _enet_address mac)
```

参数

device [in] — 设备标识

mac [in] — 指向 mac 地址结构的指针

说明

简单函数，通过提供以太网设备编号返回以太网 MAC 地址。

返回值

如果成功，返回值为 TRUE（并填入 MAC 地址），否则返回值为 FALSE。

7.64 ipcfg_get_state()

返回给定以太网设备的 IPCFG 状态。

概要

```
IPCFG_STATE ipcfg_get_state(
        uint32_t device)
```

参数

device [in] — 设备标识

说明

该函数返回以太网设备被 IPCFG 引擎评估时的当前状态。

返回值

实际的 IPCFG 状态（`enum IPCFG_STATE` 值）。

其中之一

- IPCFG_STATE_INIT
- IPCFG_STATE_UNBOUND
- IPCFG_STATE_BUSY
- IPCFG_STATE_STATIC_IP
- IPCFG_STATE_DHCP_IP
- IPCFG_STATE_AUTO_IP
- IPCFG_STATE_DHCPAUTO_IP
- IPCFG_STATE_BOOT

另请参见

- ipcfg_get_state_string()
- ipcfg_get_desired_state()

7.65 ipcfg_get_state_string()

将 IPCFG 状态值转换为字符串。

概要

```
const char *ipcfg_get_state_string(
        IPCFG_STATE state)
```

参数

state [in] — 状态标识。

说明

该函数可用来以文本消息的形式显示 IPCFG 状态值。

返回值

指向状态字符串的指针或零。

另请参见

- ipcfg_get_state()
- ipcfg_get_desired_state()

7.66 ipcfg_get_desired_state()

返回给定以太网设备的目标 IPCFG 状态。

概要

```
IPCFG_STATE ipcfg_get_desired_state(
        uint32_t device)
```

参数

device [in] — 设备标识

说明

该函数返回用户要求给定以太网设备达到的目标状态。

返回值

所需的 IPCFG 状态（`enum IPCFG_STATE` 值）。

其中之一

- `IPCFG_STATE_UNBOUND`
- `IPCFG_STATE_STATIC_IP`
- `IPCFG_STATE_DHCP_IP`
- `IPCFG_STATE_AUTO_IP`
- `IPCFG_STATE_DHCPAUTO_IP`
- `IPCFG_STATE_BOOT`

另请参见

- ipcfg_get_state_string()
- ipcfg_get_state()

7.67 ipcfg_get_link_active()

返回当前以太网链路状态

概要

```
bool ipcfg_get_link_active
        uint32_t device
```

参数

device [in] — 设备标识

说明

该函数返回给定设备的当前以太网链路状态。

返回值

如果链路为活动状态，返回值为 TRUE，否则返回值为 FALSE

另请参见

- ipcfg_get_state_string()
- ipcfg_get_state()
- ipcfg_get_desired_state()

7.68 ipcfg_get_dns_ip()

返回已注册 DNS 列表中的第 n 个 DNS IPv4 地址。

概要

```
_ip_address ipcfg_get_dns_ip(
        uint32_t device,
        uint32_t n)
```

参数

device [in] — 设备标识

n [in] — DNS IP 地址索引

说明

该函数可用于检索所有已注册给定以太网设备的 DNS IPv4 地址（通过手动方式或 DHCP 绑定过程）。

返回值

DNS IP 地址。如果第 *n* 个地址不可用，返回零。

另请参见

- ipcfg_add_dns_ip()
- ipcfg_del_dns_ip()

7.69 ipcfg_add_dns_ip()

注册以太网设备的 DNS IPv4 地址。

概要

```
bool ipcfg_add_dns_ip (
        uint32_t device,
        _ip_address address)
```

参数

device [in] — 设备标识

address [in] — 要添加的 DNS IPv4 地址

说明

该函数将 DNS IPv4 地址添加到给定以太网设备指定的列表。

返回值

如果成功，返回值为 TRUE，否则返回值为 FALSE

另请参见

- ipcfg_del_dns_ip()

7.70 ipcfg_del_dns_ip()

注销 DNS IPv4 地址。

概要

```
bool ipcfg_del_dns_ip (
        uint32_t device,
        _ip_address address)
```

参数

device [in] — 设备标识

address [in] — 要删除的 DNS IPv4 地址

说明

该函数从给定以太网设备指定的列表中删除 DNS IPv4 地址。

返回值

如果成功，返回值为 TRUE，否则返回值为 FALSE

另请参见

- ipcfg_add_dns_ip()

7.71 ipcfg_get_ip()

返回以太网设备所绑定的当前 IPv4 地址信息。

概要

```
bool ipcfg_get_ip(
     uint32_t device,
     IPCFG_IP_ADDRESS_DATA_PTR data)
```

参数

device [in] — 设备标识。

data [in] — 指针，指向 IPv4 地址信息（IP 地址、掩码和网关)。

说明

该函数返回给定以太网设备绑定的当前 IPv4 地址信息。

返回值

如果成功并且数据结构已填充，返回值为 TRUE。如果出现错误，返回值为 FALSE。

7.72 ipcfg_get_tftp_serveraddress()

返回 TFTP 服务器地址（如有)。

概要

```
_ip_address ipcfg_get_tftp_serveraddress(
        uint32_t device)
```

参数

device [in] — 设备标识。

说明

如果最后一次 BOOTP 绑定过程分配了地址，该函数返回最后一个 TFTP 服务器地址。

返回值

TFTP 服务器的 IP 地址

另请参见

- ipcfg_get_tftp_servername()
- ipcfg_get_boot_filename()

7.73 ipcfg_get_tftp_servername()

返回 TFTP 服务器名称（如有）。

概要

```
unsigned char *ipcfg_get_tftp_serveraddress(uint32_t device)
```

参数

device [in] — 设备标识。

说明

如果最后一次 DHCP 或 BOOTP 绑定过程分配了地址，该函数返回最后一个 TFTP 服务器名称。

返回值

服务器名称字符串的指针。

另请参见

- ipcfg_get_tftp_serveraddress()
- ipcfg_get_boot_filename()

7.74 ipcfg_get_boot_filename()

返回 TFTP 引导文件的名称（如有）。

概要

```
unsigned char *ipcfg_get_boot_filename(uint32_t device)
```

参数

device [in] — 设备标识。

说明

如果最后一次 DHCP 或 BOOTP 绑定过程分配了地址，该函数返回最后一个引导文件的名称。

返回值

指向引导文件名称字符串的指针。

另请参见

- ipcfg_get_tftp_serveraddress()
- ipcfg_get_tftp_servername()

7.75 ipcfg_poll_dhcp()

轮询（完成）以太网设备的 DHCP 绑定过程。

概要

```
uint32_t ipcfg_poll_dhcp(
        uint32_t device,
        bool try_auto_ip,
        IPCFG_IP_ADDRESS_DATA_PTR auto_ip_data)
```

参数

device [in] — 设备标识。

try_auto_ip [in] — 如果 DHCP 绑定失败，尝试使用 auto-ip 自动分配地址。

auto_ip_data [in] — DHCP 绑定失败时将供使用的 Ip、掩码和网关地址信息。

说明

请参见 ipcfg_bind_dhcp() 。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER

- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

另请参见

- ipcfg_bind_dhcp()

7.76 ipcfg_task_create()

创建并启动 IPCFG 以太网链路状态监视任务。

概要

```
uint32_t ipcfg_task_create(
        uint32_t priority,
         uint32_t task_period_ms)
```

参数

priority [in] — 任务优先级。

task_period_ms [in] — 任务轮询周期，以毫秒计。

说明

链路状态任务周期性检查每个初始化以太网设备的以太网链路状态。如果链路丢失，任务自动解除该接口的绑定。当链路恢复正常时，该任务尝试使用上一次成功绑定操作的信息将接口绑定到网络。

如果设备通过调用 ipcfg_unbind() 解除了绑定，该任务将保留接口为未绑定状态。

另一种无需创建单独任务监视以太网链路状态的方式是在用户任务中周期性调用 ipcfg_task_poll() 。

返回值

- MQX_OK（成功）
- MQX_DUPLICATE_TASK_TEMPLATE_INDEX
- MQX_INVALID_TASK_ID

另请参见

- ipcfg_task_destroy()

- ipcfg_task_status()
- ipcfg_task_poll()

示例

```
void main(uint32_t param)
{
  setup_network();
  ipcfg_task_create(8, 1000);
  if (! ipcfg_task_stats()) _task_block();

  ...
  ipcfg_task_destroy(TRUE);
  while (1)
  {

    _time_delay(1000);
    ipcfg_task_poll();
  }
}
```

7.77 ipcfg_task_destroy()

向 IPCFG 任务发出退出请求信号。

概要

```
void ipcfg_task_destroy(
         bool wait_task_finish)
```

参数

wait_task_finish [in] — 如果为 TRUE，表示等待任务退出。

说明

该函数设置了一个内部标志，每次以太网链路状态监视任务时都会检查此标志。任务完成当前操作后即退出。

根据参数，该函数可能会等待任务销毁。

返回值

无

另请参见

- ipcfg_task_create()
- ipcfg_task_status()
- ipcfg_task_poll()

示例

请参见 ipcfg_task_create() 。

7.78 ipcfg_task_status()

检查 IPCFG 以太网链路状态监视任务是否正在运行。

概要

```
bool ipcfg_task_status(void)
```

说明

如果 IPCFG 以太网链路状态监视任务当前正在运行，该函数返回 TRUE，否则返回 FALSE。

返回值

如果任务正在运行，返回值为 TRUE。

如果任务未运行，返回值为 FALSE。

另请参见

- ipcfg_task_create()
- ipcfg_task_destroy()
- ipcfg_task_poll()

示例

请参见 ipcfg_task_create() 。

7.79 ipcfg_task_poll()

IPCFG 以太网链路状态监视任务的一个步骤。

概要

```
bool ipcfg_task_poll(void)
```

说明

该函数执行以太网链路状态监视任务的一个步骤。该函数可在任何用户任务中周期性调用，以模拟任务操作。在此情况下不需要创建任务本身。

返回值

TRUE，如果当前绑定过程已完成（稳定状态）。

FALSE，如果任务正处于绑定操作中（应再次调用函数）。

另请参见

- ipcfg_task_create()
- ipcfg_task_destroy()
- ipcfg_task_status()

示例

请参见 ipcfg_task_create() 。

7.80 ipcfg_unbind()

将以太网设备从网络中解绑。

概要

```
uint32_t ipcfg_unbind(
        uint32_t device)
```

参数

device [in] — 设备标识。

说明

该函数释放给定以太网设备绑定的 IPv4 地址信息。这是一个阻塞函数，也就是直到进程结束或出现错误时才会返回。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT

另请参见

- ipcfg_bind_dhcp()

示例

```
void main(uint32_t param)
{
  setup_network();

  ...

  ipcfg_unbind();
  while (1) {};
}
```

7.81 ipcfg6_bind_addr()

将 IPv6 地址信息绑定到以太网设备。

概要

```
    uint32_t ipcfg6_bind_addr(
        uint32_t                    device,
        IPCFG6_BIND_ADDR_DATA_PTR ip_data)
```

参数

device [in] — 设备标识。

ip_data[in] — 指向绑定 ip 数据结构的指针。

说明

该函数尝试使用给定的 IPv6 地址数据信息将设备绑定到网络。如果地址已使用, 将返回一个错误。这是一个阻塞函数，也就是直到进程结束或出现错误时才会返回。绑定过程中出现任何故障都会导致网络接口处于未绑定状态。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT
- RTCSERR_IPCFG_BIND

示例

相关示例请参见 shell/source/rtcs/sh_ipconfig.c，Shell_ipconfig_staticip()。

7.82 ipcfg6_unbind_addr()

将 IPv6 地址从以太网设备解绑。

概要

```
uint32_t ipcfg6_unbind_addr(
uint32_t                       device,
IPCFG6_UNBIND_ADDR_DATA_PTR    ip_data)
```

参数

device [in] — 设备标识。

ip_data[in] — 指向解绑 ip 数据结构的指针。

说明

该函数释放给定以太网设备绑定的 IPv6 地址信息。这是一个阻塞函数，也就是直到进程结束或出现错误时才会返回。

返回值

- IPCFG_OK（成功）
- RTCSERR_IPCFG_BUSY
- RTCSERR_IPCFG_DEVICE_NUMBER
- RTCSERR_IPCFG_INIT

示例

相关示例请参见 shell/source/rtcs/sh_ipconfig.c，Shell_ipconfig_unbind6()。

7.83 ipcfg6_get_addr()

返回以太网设备所绑定的 IPv6 地址信息。

概要

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

参数

device [in] — 设备标识

n [in] — 要检索的 IPv6 地址序列号（从 0 开始）。

data [in/out] — 指向 IPv6 地址信息结构（IPv6 地址、地址状态和类型）的指针。

说明

该函数返回给定以太网设备所绑定的（通过手动方式或 IPv6 无状态自动配置程序）IPv6 地址信息。

一个接口可能具有多个绑定的 IPv6 地址。

返回值

- RTCS_OK（成功，数据已填充）
- RTCS_ERROR（失败，不存在第 n 个地址）

另请参见

- ipcfg6_unbind_addr()

示例

```
/* Print all bound IPv6 addresses.*/
{
    IPCFG6_GET_ADDR_DATA     addr_data;
    char                     addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int                      n;
    for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
    {
        /* Convert IPv6 address to string presentation and print it.*/
        if(inet_ntop(AF_INET6, &addr_data.ip_addr, addr_str, sizeof(addr_str)))
        {
            printf("IP6[%d] : %s\n", n, addr_str);
        }
    }
}
```

7.84 ipcfg6_get_dns_ip()

返回以太网设备的已注册 DNS 列表中的第 n 个 DNS IPv6 地址。

概要

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

参数

device [in] — 设备标识。

n [in] — 要检索的 IPv6 地址序列号（从 0 开始）。

data [in/out] — 指向 IPv6 地址信息结构（IPv6 地址、地址状态和类型）的指针。

说明

该函数返回给定以太网设备所绑定的（通过手动方式或 IPv6 无状态自动配置程序）IPv6 地址信息。

一个接口可能具有多个绑定的 IPv6 地址。

返回值

- RTCS_OK（成功，数据已填充）
- RTCS_ERROR（失败，不存在第 n 个地址）

另请参见

- ipcfg6_unbind_addr()

示例

```
/* Print all bound IPv6 addresses.*/
{
    IPCFG6_GET_ADDR_DATA     addr_data;
    char                     addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int                      n;
    for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
    {
       /* Convert IPv6 address to string presentation and print it.*/
       if(inet_ntop(AF_INET6, &addr_data.ip_addr, addr_str, sizeof(addr_str)))
       {
          printf("IP6[%d] : %s\n", n, addr_str);
       }
    }
}
```

7.85 ipcfg6_add_dns_ip()

注册以太网设备的 DNS IPv6 地址。

概要

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

参数

device [in] — 设备标识。

n [in] — 要检索的 IPv6 地址序号（从 0 开始）。

说明

该函数将 DNS IPv6 地址添加到给定以太网设备指定的列表。

返回值

如果成功，返回值为 TRUE，否则返回值为 FALSE

另请参见

- ipcfg6_get_dns_ip()
- ipcfg6_del_dns_ip()

示例

```
/* Register DNS IPv6 address with the Ethernet device.*/
    {
       char              *addr_str = "2001:470:1234:567:4c39:64fa:1caa:44c8";
       in6_addr          dns6_addr;

       if(inet_pton(AF_INET6, addr_str, &dns6_addr, sizeof(dns6_addr)) == RTCS_OK)
       {
          if(ipcfg6_add_dns_ip(BSP_DEFAULT_ENET_DEVICE, &dns6_addr) == TRUE)
          {
              printf("Adding DNS address is successful.\n");
          }
          else
          {
              printf("Adding DNS address is failed.\n");
          }
        }
    }
```

7.86 ipcfg6_del_dns_ip()

返回以太网设备所绑定的 IPv6 地址信息。

概要

```
uint32_t ipcfg6_get_addr(uint32_t device, uint32_t n, IPCFG6_GET_ADDR_DATA_PTR data)
```

参数

device [in] — 设备标识。

dns_addr [in] — 要删除的 DNS IPv6 地址。

说明

该函数从给定以太网设备指定的列表中删除 DNS IPv6 地址。

返回值

如果成功，返回值为 TRUE，否则返回值为 FALSE

另请参见

- ipcfg6_get_dns_ip()
- ipcfg6_add_dns_ip()

示例

```
/* Print all bound IPv6 addresses.*/
{
   IPCFG6_GET_ADDR_DATA    addr_data;
   char                    addr_str[RTCS_IP6_ADDR_STR_SIZE];
   int                     n;

   for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
   {
```

7.87 ipcfg6_get_scope_id()

返回以太网设备所绑定的 IPv6 地址信息。

概要

```
uint32_t ipcfg6_get_scope_id (uint32_t device)
```

参数

device [in] — 设备标识。

说明

该函数返回为以太网设备分配的范围 ID（接口标识符）。

范围 ID 用于指示发送/接收通信流所使用的网络接口。

返回值

- 范围 ID（成功）
- 0（失败）

示例

```
/* Print all bound IPv6 addresses.*/
{
   IPCFG6_GET_ADDR_DATA    addr_data;
   char                    addr_str[RTCS_IP6_ADDR_STR_SIZE];
   int                     n;

   for(n=0;(ipcfg6_get_addr(BSP_DEFAULT_ENET_DEVICE, n, &addr_data) == RTCS_OK); n++)
   {
```

7.88 iwcfg_set_essid()

概要

```
     uint32_t iwcfg_set_essid
          (
             uint32_t dev_num,
```

```
        char *essid
    )
```

参数

dev_num [in] — 设备标识（索引）。

essid [in] — ESSID（扩展服务集标识符）字符串的指针。

说明

该函数设定设备标识的 IP 接口结构 ESSID。设备必须先进行初始化。只有当用户确认其修改后，ESSID 才变为有效。ESSID 用于识别属于相同虚拟网络中的单元。

返回值

- ENET_OK（成功）
- ENET_ERROR
- ENETERR_INVALID_DEVICE

示例

```
#define SSID           "NGZG"
#define DEFAULT_DEVICE 1
int32_t                 error;
/* IP configuration */
error = RTCS_create();
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address);
error = ipcfg_init_device (DEFAULT_DEVICE, enet_address);
/* Set SSID */
iwcfg_set_essid (DEFAULT_DEVICE, SSID);
iwcfg_commit( DEFAULT_DEVICE );
/* end of IP configuration */
error = ipcfg_bind_staticip (DEFAULT_DEVICE, &ip_data);
```

7.89 iwcfg_get_essid()

概要

```
uint32_t iwcfg_get_essid
    (
        uint32_t dev_num,
        char *essid
    )
```

参数

dev_num [in] — 设备标识（索引）。

essid [out] — 扩展服务集标识符字符串。

说明

该函数返回用于所选设备的 ESSID。

返回值

- ENET_OK（成功）
- ENET_ERROR
- ENETERR_INVALID_DEVICE

示例

```
#define DEFAULT_DEVICE 1
char[20] ssid_name;
iwcfg_get_ssid (DEFAULT_DEVICE, &ssid_name);
```

7.90 iwcfg_commit()

概要

```
    uint32_t iwcfg_commit
        (
            uint32_t dev_num
        )
```

参数

dev_num [in] — 设备标识（索引）。

说明

提交所请求的更改。某些板卡可能无法立即应用完成的更改（可能会等待更改汇集）。该命令强制板卡应用所有挂起的更改。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE
- 其他设备特定错误

示例

```
#define SSID           "NGZG"
#define DEFAULT_DEVICE 1
/* initialize rtcs before */
iwcfg_set_essid (DEFAULT_DEVICE, SSID);
iwcfg_commit (DEFAULT_DEVICE);
```

7.91 iwcfg_set_mode()

概要

```
uint32_t iwcfg_set_mode
    (
       uint32_t dev_num,
       char *mode
    )
```

参数

dev_num [in] — 设备标识（索引）。

mode [in] — WiFi 设备模式，可接受的值为“managed”和“adhoc”。

说明

根据网络拓扑设定设备的工作模式。模式可以为“Ad-Hoc”，意味着网络只由一个单元组成，没有接入点；或者为“managed”，说明这是一个连接到由许多接入点组成的带漫游功能的网络中的节点。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE
- 其他设备特定错误

示例

```
#define DEMOCFG_SECURITY "none"
#define DEMOCFG_SSID "NGZG"
#define DEMOCFG_NW_MODE "managed"
#define DEFAULT_DEVICE 1
error = RTCS_create();
ip_data.ip = ENET_IPADDR;
ip_data.mask = ENET_IPMASK;
ip_data.gateway = ENET_IPGATEWAY;
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address); error = ipcfg_init_device
(DEFAULT_DEVICE, enet_address);
iwcfg_set_essid (DEFAULT_DEVICE, DEMOCFG_SSID );
iwcfg_set_sec_type (DEFAULT_DEVICE, DEMOCFG_SECURITY);
iwcfg_set_mode (DEFAULT_DEVICE, DEMOCFG_NW_MODE);
error = ipcfg_bind_staticip (DEFAULT_DEVICE, &ip_data);
```

7.92 iwcfg_get_mode()

概要

```
uint32_t iwcfg_get_mode
    (
        uint32_t dev_num
        char *mode
    )
```

参数

dev_num [in] — 设备标识（索引）。

mode [out] — 当前 wifi 模式（字符串）。

说明

返回当前 wifi 模块的模式。可能的值为“managed” 或“adhoc”。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

示例

```
#define DEFAULT_DEVICE 1
char[20] ssid_name;
iwcfg_get_mode (DEFAULT_DEVICE, &ssid_name);
```

7.93 iwcfg_set_wep_key()

概要

```
uint32_t iwcfg_set_wep_key
    (
        uint32_t dev_num,
        char     *wep_key,
        uint32_t key_len,
        uint32_t key_index
    )
```

参数

dev_num [in] — 设备标识（索引）。

wep_key [in] — *Wep_key*。

key_len [in] — 密钥长度。

key_index [in] — 其他可选的设备特定参数。索引必须低于 256。

说明

设定 WiFi 设备的 wep 密钥。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

示例

```
iwcfg_set_wep_key (DEFAULT_DEVICE, DEMOCFG_WEP_KEY, strlen(DEMOCFG_WEP_KEY),
DEMOCFG_WEP_KEY_INDEX);
```

7.94 iwcfg_get_wep_key()

概要

```
    uint32_t iwcfg_get_wep_key
        (
            uint32_t dev_num,
            char     *wep_key,
            uint32_t key_index
        )
```

参数

dev_num [in] — 设备标识（索引）。

wep_key [in] — Wep_key。

key_index [in] — 其他可选的设备特定参数。索引必须低于 256。

说明

获取 wep 密钥。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

7.95 iwcfg_set_passphrase()

概要

```
    uint32_t iwcfg_set_passphrase
        (
           uint32_t   dev_num,
           char     *passphrase
        )
```

参数

dev_num [in] — 设备标识（索引）。

passphrase [in] — SSID 密码。

说明

设定 wpa 密码。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

示例

```
#define DEMOCFG_SECURITY "wpa"
#define DEMOCFG_SSID "NGZG"
#define DEMOCFG_NW_MODE "managed"
#define DEMOCFG_PASSPHRASE "abcdefgh"
#define DEFAULT_DEVICE 1
error = RTCS_create();
ip_data.ip = ENET_IPADDR;
ip_data.mask = ENET_IPMASK;
ip_data.gateway = ENET_IPGATEWAY;
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address) error = ipcfg_init_device
(DEFAULT_DEVICE, enet_address);
iwcfg_set_essid (DEFAULT_DEVICE, DEMOCFG_SSID);
iwcfg_set_passphrase (DEFAULT_DEVICE, DEMOCFG_PASSPHRASE);
iwcfg_set_sec_type (DEFAULT_DEVICE, DEMOCFG_SECURITY);
iwcfg_set_mode (DEFAULT_DEVICE, DEMOCFG_NW_MODE);
error = ipcfg_bind_staticip (DEFAULT_DEVICE, &ip_data);
```

7.96 iwcfg_get_passphrase()

概要

```
    uint32_t iwcfg_get_passphrase
         (
            uint32_t dev_num,
            char *passphrase
         )
```

参数

dev_num [in] — 设备标识（索引）。

passphrase [out] — SSID 密码（字符串）。

说明

从初始化的 wifi 设备获取 wpa 密码。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

7.97 iwcfg_set_sec_type()

概要

```
uint32_t iwcfg_set_sec_type
    (
        uint32_t dev_num,
        char     *sec_type
    )
```

参数

dev_num [in] — 设备标识（索引）。

sec_type [in] — 安全类型。可接受的值为"none"、"wep"、"wpa"和"wpa2"。

说明

设定设备的安全类型。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

示例

参见 iwcfg_set_passphrase 示例。

7.98 iwcfg_get_sectype()

概要

```
uint32_t iwcfg_get_sec_type
    (
        uint32_t dev_num,
        char *sec_type
    )
```

参数

dev_num [in] — 设备标识（索引）。

sec_type [out] — 安全类型（字符串）。

说明

获取设备的安全类型。可能值为“none”、“wep”、“wpa”和“wpa2”。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

7.99 iwcfg_set_power()

概要

```
uint32_t iwcfg_set_power
    (
        uint32_t dev_num,
        uint32_t pow_val,
        uint32_t flags
    )
```

参数

dev_num [in] — 设备标识（索引）。

pow_val [in] — 功率以 dBm 为单位。

flags [in] — 设备特定选择。

说明

设定网卡的传输功率 (以 dBm 为单位) 以支持多种传输功率。假设 W 为以 Watt 为单位的功率，则以 dBm 为单位的功率 P = 30 + 10.log(W)。

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

7.100 iwcfg_set_scan()

概要

```
uint32_t iwcfg_set_scan
    (
        uint32_t dev_num,
```

```
            char *ssid
        )
```

参数

dev_num [in] — 设备标识（索引）。

ssid [in] — 尚未使用。

说明

该函数将搜索所有可用网络并打印格式。该格式取决于 WiFi 供应商。

ssid = tplink - SSID 名称

bssid = 94:c:6d:a5:51:b - SSID 的 MAC 地址

channel = 1 - 通道

strength = ##### - 图形化的信号强度

indicator = 183 - 信号强度

返回值

- ENET_OK（成功）
- ENETERR_INVALID_DEVICE

示例

```
#define SSID    "NGZG"
int32_t         error;
/* IP configuration */
error = RTCS_create();
ENET_get_mac_address (DEFAULT_DEVICE, ENET_IPADDR, enet_address);
error = ipcfg_init_device (DEFAULT_DEVICE, ENET_IPADDR);
/* scan for networks */
iwcfg_set_scan (DEFAULT_DEVICE, NULL);
Example output:
ssid = tplink
bssid = 94:c:6d:a5:51:b
channel = 1
strength = #####
indicator = 183
ssid = Faz
bssid = 0:21:91:12:da:cc
channel = 1
strength = ####.
indicator = 172
---
scan done.
```

7.101 listen()

将数据流套接字置为监听状态。

概要

```
uint32_t listen(
        uint32_t socket,
        uint16_t backlog)
```

参数

socket [in] — 套接字句柄

backlog [in] — 为 RTCS 提供的提示，用于确定每个监听套接字的最大待创建连接数。

说明

将数据流置于监听状态允许来自远程端点的接入连接请求。在应用程序调用 listen()后，其应调用 accept()以将新的套接字连接到接入请求。

该函数阻塞，但是会立即执行并回复命令。第二个函数参数“backlog”有助于 RTCS 确定每个监听套接字的最大等待中的已创建连接数。已创建连接是处理三步握手（SYN、SYN/ACK 和 ACK）模式的连接。等待中的已创建连接是应用程序还未调用 accept()的已创建连接。backlog 参数小于零创建了零长度的 backlog 队列，直接导致丢弃该套接字的所有连接请求。对于大于等于零的 backlog 参数，backlog + 1 指定了最大等待中的已创建连接数；除非该参数大于 RTCSCFG_SOMAXCONN，这种情况下 RTCSCFG_SOMAXCONN 指定了最大等待中的已创建连接数。如果 backlog 队列已满且从同一个监听套接字中接收了新的连接请求，RTCS 将丢弃该连接请求。调用 accept()函数将使 backlog 队列空出空间。可在运行时修改监听套接字的 backlog 队列长度，通过调用 listen()函数，以及监听套接字句柄参数和 backlog 参数新值实现。对于允许的最大连接数（已创建和半开放式）有系统级的配置限制 RTCSCFG_TCP_MAX_CONNECTIONS。一个非零正值限制了 TCB 的数量，所有后台创建的监听套接字对连接要求作出回应。零（默认）不存在限制。如果系统级的 RTCSCFG_TCP_MAX_HALF_OPEN 设定为非零正值，则 RTCS 试图监视半开放连接，并且当接收到新的连接请求时丢弃旧的半开放连接。

返回值

- RTCS_OK（成功）
- 特定错误代码（失败）

另请参见

- accept()
- bind()
- socket()

示例

请参见 accept() 。

7.102 MIB1213_init()

初始化 MIB-1213。

概要

```
void MIB1213_init(void)
```

说明

该函数安装了在 RFC 1213 中定义的标准 MIB。如果未调用该函数，SNMP 代理将无法访问 MIB。

另请参见

- SNMP_init()

示例

请参见 SNMP_init() 。

7.103 MIB_find_objectname()

寻找表中的对象。

概要

```
bool MIB_find_objectname(uint32_t op, void *index, void * *instance)
```

参数

op [in]

index [in] — 指向包含表索引的结构的指针。

instance [out]

说明

对于表中的每个变量对象，您必须提供 MIB_find_objectname()，其中 objectname 为变量对象的名称。该函数获取一个实例指针。

返回值

- SNMP_ERROR_noError（成功）
- SNMP_ERROR_wrongValue
- SNMP_ERROR_inconsistentValue
- SNMP_ERROR_wrongLength
- SNMP_ERROR_resourceUnavailable
- SNMP_ERROR_genErr

另请参见

- SNMP_init()
- MIB1213_init()

示例

7.104 MIB_set_objectname()

为表中的可写对象设置名称。

概要

```
uint32_t MIB_set_objectname(void *instance, unsigned char *value_ptr, uint32_t value_len)
```

参数

instance [in]

value_ptr [out] — 用于设置 objectname 值的指针。

value_len [out] — 值的长度（以字节为单位）。

说明

对于每个可写变量对象，您必须提供 MIB_set_objectname()，其中 objectname 为变量对象的名称。

另请参见

- SNMP_init()
- MIB1213_init()
- MIB_find_objectname()

示例

7.105 NAT_close()

停止网络地址转换。

概要

```
uint32_t  NAT_close(void)
```

返回值

- RTCS_OK（成功）

另请参见

- NAT_init()

7.106 NAT_init()

启动网络地址转换。

概要

```
uint32_t NAT_init(
        _ip_address prv_network,
        _ip_address prv_netmask)
```

参数

prv_network [in] — 专用网络地址

prv_netmask [in] — 专用网络子网掩码

说明

只有当网络地址转换启动后，Freescale MQX NAT 才开始工作，这通过调用 NAT_init()，并且_IP_forward 全局参数为真时实现。

函数 NAT_init()使能了在 NAT_alg_table 中定义的所有应用级的网关。欲了解更多信息，请参见禁用 NAT 应用层网关 。

在您调用 **NAT_close()**后，可以使用该函数重启网络地址转换。

返回值

- RTCS_OK（成功）

- RTCSERR_OUT_OF_MEMORY（失败）
- RTCSERR_INVALID_PARAMETER（失败）

另请参见

- NAT_close()
- NAT_stats()
- nat_ports
- nat_timeouts
- NAT_STATS

7.107 NAT_stats()

获取网络地址转换统计数据。

概要

```
NAT_STATS_PTR  NAT_stats(void)
```

返回值

- 指向 NAT_STATS 结构的指针（成功）
- NULL（失败：未调用 NAT_init()）

另请参见

- NAT_init()
- NAT_STATS

7.108 ping()

请参见 RTCS_ping() 。

7.109 PPP_init()

初始化 PPP 链接的 PPP 驱动程序。

概要

```
_ppp_handle PPP_init(
        PPP_PARAM_STRUCT* params
)
```

参数

params[in/out] — 用于 PPP 初始化的参数。这里存放由 PPP 创建的 IPCP 句柄。

说明

如果 RTCS 不能执行以下任一操作，函数 PPP_initialize()将失败:

- 打开低层设备（即，“ittyd:”）。
- 初始化 HDLC 层。
- 初始化 LCP 层。
- 分配消息池。
- 创建接收和发送任务。
- 打开 HDLC 层。
- 添加 PPP 接口。
- 在 IPCP 层上绑定 IP 地址。

返回值

- PPP 设备句柄。
- 零。

另请参见

- PPP_release
- PPP_pause
- PPP_resume
- PPP_PARAM_STRUCT

示例

```
/* Start PPP in listen mode */
{
        PPP_PARAM_STRUCT params;
        uint32_t handle;

        mem_zero(&params, sizeof(params));
```

```
        params.device = "ittyd:";
        /* Set local IP address to 192.168.1.201 */
        params.local_addr = 0xC0A801C9;
        /* Set remote IP address to 192.168.1.202 */
        params.remote_addr = 0xC0A801CA;
        params.listen_flag = 1;
        /* Init PPP */
        handle = PPP_init(&params);
        if (handle == NULL)
        {
            fprintf(stderr,"PPP initialization failed.");
        }
        else
        {
            PPP_pause(handle);
            /* Do something on ittyd: device here */
            PPP_resume(handle);
            if (PPP_release(ppp_conn->PPP_HANDLE) != RTCS_OK)
        {
            fprintf(stderr,"Failed to release PPP connection.");
            }
        }
}
```

7.110 PPP_release()

取消对 PPP 驱动程序的初始化并释放低层设备。

概要

```
uint32_t PPP_release(
_ppp_handle handle
)
```

参数

handle[in]— PPP 设备的句柄。

说明

该函数用于释放 PPP 设备使用的所有资源。其执行以下步骤:

- 在 IPCP 层上取消 IP 地址的绑定。
- 终止 PPP 的内部接收和发送任务。
- 关闭 HDLC 层。
- 关闭 LCP 层。
- 取消对于消息池的分配。
- 关闭低层设备。
- 移除 PPP 接口。
- 释放内存。

返回值

- RTCS_OK，如果释放成功。
- 错误代码。

另请参见

- PPP_init()

示例

请参见 PPP_init()作为示例。

7.111 PPP_pause()

暂停 PPP 状态机，以便低层设备可用于其他通信。

概要

```
uint32_t PPP_pause(
_ppp_handle handle
```

参数

handle[in] — 用于要暂停的 PPP 设备的句柄。

说明

当 PPP 暂停时，所有具有远程对等端的通信将停止，低层设备可做其他用途。

这一般包括向 GPRS 调制解调器发送 AT 命令，以及与运行 Windows 操作系统的机器进行握手。

返回值

- RTCS_OK，如果成功。
- 错误代码。

另请参见

- PPP_resume()

示例

参见 PPP_init()作为示例。

7.112 PPP_resume()

恢复 PPP 状态机。

概要

```
uint32_t PPP_resume(
_ppp_handle handle
)
```

参数

handle[in] — 用于要恢复的 PPP 设备的句柄。

说明

该函数用于恢复 PPP 链接上的通信，并与 PPP_pause 函数配合使用。

返回值

- RTCS_OK，如果成功。
- 错误代码。

另请参见

- PPP_pause()

示例

参见 PPP_init()作为示例。

7.113 recv()

为 RTCS 提供输入缓冲区。

概要

```
int32_t recv(
        uint32_t    socket,
        char *      buffer,
        uint32_t    buflen,
        uint32_t    flags
)
```

参数

socket [in] — 用于已连接的数据流套接字的句柄。

buffer [out] — 指向放置已接收数据的缓冲区的指针。

buflen [in] — 缓冲区大小（以字节为单位）。

flags [in] — 基础协议的标志。例如:

RTCS_MSG_PEEK — 用于 UDP 套接字。接收数据报但不使用（忽略数据流套接字）。

MSG_DONTWAIT — 用于 TCP 套接字。该函数调用作为无阻塞执行，应用了接收和推送套接字选项。

MSG_WAITALL — 用于 TCP 套接字。该函数调用作为阻塞执行。忽略接收到的 TCP 推送标志，当接收了足够的数据用于填充缓冲区时，将返回 recv()函数。如果发生超时、错误或断开连接从而导致少于所需数据时，仍会返回 recv()函数。

Zero — 忽略。

说明

函数 recv()为 RTCS 提供用于在数据流或数据报套接字上进行数据输入的缓冲区。

当标志参数为 RTCS_MSG_PEEK 时，在下次调用 recv()或 recvfrom()时会接收相同的数据报。

如果函数返回 RTCS_ERROR, 应用程序可以调用 RTCS_geterror() 来确定错误原因。

注

> 如果对等端成功关闭连接，则 recv()返回 RTCS_ERROR，而不是 BSD 4.4 中指定的 0。后续调用 RTCS_geterror()将返回 RTCSERR_TCP_CONN_CLOSING。

数据流套接字

如果接收无等待套接字选项为 TRUE，则 RTCS 会立即复制内部缓冲数据（最多 buflen 字节）到缓冲区（buffer 参数的缓冲区），且 recv()返回。如果接收无等待套接字选项为 FALSE，则 recv()阻塞直到缓冲区满或满足接收推送套接字选项。

如果接收推送套接字选项为 TRUE，无论数据是否接收，接收 TCP 推送标志均会导致 recv()返回。如果接收推送套接字选项为 FALSE，RTCS 会忽略输入的 TCP 的 push 标志，当接收了足够的数据用于填充缓冲区时，recv()才会返回。

数据报套接字

数据报套接字上的 recv()函数与 recvfrom()具有相同的 NULL fromaddr 和 fromlen 指针。recv()函数一般用于连接套接字上。

数据流套接字

```
uint32_t handle;
char buffer[20000];
uint32_t count;

...
count = recv(handle, buffer, 20000, 0);
if (count == RTCS_ERROR)
{
  printf("\nError, recv() failed with error code %lx",
         RTCS_geterror(handle));
} else {
  printf("\nReceived %ld bytes of data.", count);
}
```

7.114 recvfrom()

为 RTCS 提供存放输入到数据报套接字的数据的缓冲区。

概要

```
int32_t recvfrom(
        uint32_t      socket,
        char *        buffer,
        uint32_t      buflen,
        uint32_t      flags,
        sockaddr *   fromaddr,
        uint16_t     *fromlen)
```

参数

socket [in] — 用于数据报套接字的句柄。

buffer [out] — 指向存放已接收数据的缓冲区的指针。

buflen [in] — 缓冲区大小（以字节为单位）。

flags [in] — 基础协议的标志。以下之一:

RTCS_MSG_PEEK — 接收数据报但不使用。

零 — 忽略。

fromaddr [out] — 消息的源套接字地址。

fromlen [in/out] — 当输入时：fromaddr 缓冲区大小。

当输出时：存放在 fromaddr 缓冲区中的套接字地址大小，或者如果所提供的缓冲区太小（套接字地址被截短），则为截短前的长度。

说明

如果在 connect()中指定远程端点，将只从该源接收数据报。

当标志参数为 RTCS_MSG_PEEK 时，在下次调用 recv() 或 recvfrom() 时会接收相同的数据报。

如果 fromlen 为 NULL，套接字地址不会写入 fromaddr。如果 fromaddr 为 NULL 且 fromlen 的值不为 NULL，则结果不确定。

如果函数返回 RTCS_ERROR，应用程序可以调用 RTCS_geterror()来确定错误原因。

该函数阻塞直到数据可用或发生错误为止。

返回值

- 接收的字节数（成功）
- RTCS_ERROR（失败）

另请参见

- bind()
- RTCS_geterror()
- sendto()
- socket()

示例

最多接收 500 字节的数据。

```
uint32_t       handle;
sockaddr_in    remote_sin;
uint32_t        count;
char           my_buffer[500];
uint16_t         remote_len = sizeof(remote_sin);

...

count = recvfrom(handle, my_buffer, 500, 0, (struct sockaddr *) &remote_sin,
&remote_len);

if (count == RTCS_ERROR)
{
   printf("\nrecvfrom() failed with error %lx",
         RTCS_geterror(handle));
} else {
  printf("\nReceived %ld bytes of data.", count);
}
```

7.115 RTCS_attachsock()

接管套接字的所有权。

概要

```
uint32_t RTCS_attachsock(
         uint32_t socket)
```

参数

socket [in] — 套接字句柄。

说明

该函数向套接字的拥有者列表添加调用任务。

该函数阻塞，但是会立即执行并响应命令。

返回值

- 新套接字句柄（成功）
- RTCS_SOCKET_ERROR（失败）

另请参见

- accept()
- RTCS_detachsock()

示例

主任务循环以接受连接。当接受连接时，主任务会创建一个子任务来管理该连接。主任务通过调用 RTCS_detachsock()来放弃对套接字的控制并创建具有已接受套接字句柄的子任务作为初始参数。

```
while (TRUE) {
  /* Issue ACCEPT: */
  TELNET_accept_skt =
   accept(TELNET_listen_skt, &peer_addr, &addr_len);
  if (TELNET_accept_skt != RTCS_SOCKET_ERROR) {
     /* Transfer the socket and create the child task to look after
        the socket: */
     if (RTCS_detachsock(TELNET_accept_skt) == RTCS_OK) {
        child_task = (_task_create(LOCAL_ID, CHILD, TELNET_accept_skt);
    } else {
      printf("\naccept() failed, error
              0x%lx", RTCS_geterror(TELNET_accept_skt));
    }
  }
}
The child attaches itself to the socket for which the main task transferred ownership.
void TELNET_Child_task
 (
   uint32_t socket_handle
 )
 {
   /* Attach the socket to this task: */
   printf("\nCHILD - about to attach the socket.");
   socket_handle = RTCS_attachsock(socket_handle);
   if (socket_handle != RTCS_SOCKET_ERROR) {
     /* Continue managing the socket. */
   } else {
   ...
```

7.116 RTCS_create()

创建 RTCS。

概要

```
uint32_t RTCS_create(void)
```

说明

该函数分配 RTCS 所需资源并创建 TCP/IP 任务。

返回值

- RTCS_OK （成功）
- 错误代码 （失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()

示例

请参见示例：设置 RTCS 。

7.117 RTCS_detachsock()

放弃套接字的所有权。

概要

```
    uint32_t RTCS_detachsock(
            uint32_t socket)
```

参数

socket [in] — 套接字句柄

说明

该函数将调用的任务从用户的套接字列表中移除。

参数套接字返回以下之一：

- socket()
- accept()
- RTCS_attachsock()

虽然命令立刻执行，但是该函数仍然阻塞。

返回值

- RTCS_OK （成功）
- 特定错误代码 （失败）

另请参见

- accept()
- RTCS_attachsock()
- socket()

示例

请参见 RTCS_attachsock() 。

7.118 RTCS_gate_add()

向 RTCS 添加网关。

概要

```
uint32_t RTCS_gate_add(
        _ip_address gateway,
        _ip_address network,
        _ip_address netmask)
```

参数

gateway [in] — 网关的 IP 地址。

network [in] — 网关所在的 IP 网络。

netmask [in] — 网络的网络掩码。

说明

函数 RTCS_gate_add()向 RTCS 添加路由跳数为零的网关。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_gate_remove()
- RTCS_if_bind*函数系列

示例

添加默认网关。

```
error = RTCS_gate_add(GATE_ADDR, INADDR_ANY, INADDR_ANY);
```

7.119 RTCS_gate_add_metric()

向 RTCS 路由表添加网关并指定其路由跳数。

概要

```
    uint32_t RTCS_gate_add_metric(
            _ip_address gateway,
            _ip_address network,
            _ip_address netmask
            _uint16_t metric)
```

参数

gateway [in] — 网关的 IP 地址。

network [in] — 网关所在的 IP 网络。

netmask [in] — 网络的网络掩码。

metric [in] — 网关路由跳数的范围为 0 至 65535。

说明

函数 RTCS_gate_add_metric()指定网关的路由跳数。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_gate_remove_metric()
- RTCS_if_bind*函数系列

示例

```
RTCS_gate_add_metric(GATE_ADDR, INADDR_ANY, INADDR_ANY, 42)
```

7.120 RTCS_gate_remove()

从路由表删除网关。

概要

```
uint32_t RTCS_gate_remove(
        _ip_address gateway,
        _ip_address network,
        _ip_address netmask)
```

参数

gateway [in] — 网关的 IP 地址。

network [in] — 网关所在的 IP 网络。

netmask [in] — 网络的网络掩码。

说明

函数 RTCS_gate_remove()从路由表删除网关。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- RTCS_gate_add()

示例

删除默认网关。

```
error = RTCS_gate_remove(GATE_ADDR, INADDR_ANY, INADDR_ANY);
```

7.121 RTCS_gate_remove_metric()

从路由表中删除指定网关。

概要

```
uint32_t RTCS_gate_remove_metric(
        _ip_address gateway,
        _ip_address network,
        _ip_address netmask
        _uint16_t   metric)
```

参数

gateway [in] — 网关的 IP 地址

network [in] — 网关所在的 IP 网络

netmask [in] — 网络的网络掩码

metric [in] — 网关路由跳数的范围为 0 至 65535

说明

函数 RTCS_gate_remove_metric()用于从路由表中删除指定网关 (如果该网关匹配网络、网络掩码和路由跳数)。

返回值

- RTCS_OK （成功)
- 错误代码 （失败)

另请参见

- RTCS_gate_add_metric()

示例

```
error = RTCS_gate_remove_metric
                  (GATE_ADDR, INADDR_ANY, INADDR_ANY, 42)
```

7.122 RTCS_get_errno

返回并清零 RTCS_errno

概要

```
uint32_t     RTCS_get_errno(void)
```

参数

- 无参数。

说明

RTCS_errno 的返回值，当 select()或 accept()函数返回-1（RTCS_SOCKET_ERROR）时，由 RTCS 设定的任务特定错误值。可用于确定套接字是强制关闭（由 closesocket()）还是通知关闭（由 shutdownsocket()或来自已连接的远程主机的输入 TCP RST 标志）。它还将 RTCS_errno 清零。

返回值

特定错误代码

另请参见

- select()
- accept()

示例

```
uint32_t rtcserrno = RTCS_get_errno();
```

7.123 RTCS_geterror()

获取 RTCS 函数返回套接字错误的原因。

概要

```
    uint32_t RTCS_geterror(
          uint32_t socket)
```

参数

socket [in] — 套接字句柄

说明

该函数不阻塞。如果 accept()返回 RTCS_SOCKET_ERROR 且 RTCS_errno 与 RTCSERR_SOCK_CLOSED 不同，或以下任一函数返回 RTCS_ERROR 时，使用该函数：

- recv()
- recvfrom()

- send()
- sendto()

返回值

- RTCS_OK（无套接字错误）
- 上一个套接字错误代码

另请参见

- accept()
- recv()
- recvfrom()
- send()
- sendto()

示例

请参见 accept()、recv()、recvfrom()、send()和 sendto()。

7.124 RTCS_if_add()

向 RTCS 添加设备接口。

概要

```
uint32_t RTCS_if_add(
        void               *dev_handle,
        RTCS_IF_STRUCT_PTR     callback_ptr,
        _rtcs_if_handle * rtcs_if_handle)
```

参数

dev_handle [in] — **ENET_initialize()**或 **PPP_initialize()**的句柄。

callback_ptr [in] — 以下之一：

指向用于设备接口的回调函数的指针。

RTCS_IF_ENET（仅以太网：对于以太网接口使用默认的回调函数）。

RTCS_IF_LOCALHOST（对于本地环回使用默认的回调函数）。

RTCS_IF_PPP（仅 PPP：对于 PPP 接口使用默认回调函数）。

rtcs_if_handle [out] — 指向 RTCS 接口句柄的指针。

说明

应用程序使用 RTCS 接口句柄来调用 RTCS_if_bind 函数。

返回值

- **RTCS_OK** (成功)
- 错误代码 (失败)

另请参见

- ENET_initialize()
- PPP_init()
- RTCS_create()
- RTCS_if_bind()
- *RTCS_IF_STRUCT*

示例

请参见示例: 设置 RTCS 。

7.125 RTCS_if_get_addr()

返回绑定至设备接口的 IPv4 地址。

概要

```
_ip_address RTCS_if_get_addr(_rtcs_if_handle ihandle)
```

参数

- *ihandle [in]* — RTCS 接口句柄

说明

该函数用于检索绑定至给定设备接口的 IPv4 地址。

返回值

- IPv4 地址

另请参见

- RTCS_if_bind()

示例

```
/* Print IPv4 address bound to interface. */
{
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
    _ip_address     addr = RTCS_if_get_addr(ihandle);
    char            prn_addr[RTCS_IP4_ADDR_STR_SIZE];

    inet_ntop(AF_INET, &addr, prn_addr, sizeof(prn_addr))
    printf("IPv4 Interface Address: %s\n", prn_addr);
}
```

7.126 RTCS_if_get_handle ()

返回第 n 个接口的 RTCS 句柄。

概要

```
_rtcs_if_handle RTCS_if_get_handle(uint32_t n)
```

参数

n [in] — 接口索引 (从零开始)。

说明

该函数返回第 n 个接口 (从零开始) 的句柄。如果不存在第 n 个接口，则返回 0。

返回值

- RTCS 接口句柄
- 0 (如果第 n 个接口不存在)

另请参见

- RTCS_if_add()

示例

```
/* Print number of registered interfaces.*/
{
   int i = 0;

   while(RTCS_if_get_handle(i)!= 0)
     {
       i++;
     }

     printf("There are %d registered interfaces.\n", i);
}
```

7.127 RTCS_if_get_mtu()

概要

```
uint32_t RTCS_if_get_mtu(_rtcs_if_handle ihandle)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

说明

该函数返回与 *rtcs_if_handle* 相关的设备接口的最大传输单元（MTU)。

返回值

- 最大传输单元（成功)
- 0 （失败)

另请参见

- RTCS_if_add()

7.128 RTCS_if_bind()

将 IP 地址和网络掩码绑定到设备接口。

概要

```
uint32_t RTCS_if_bind(
         _rtcs_if_handle   rtcs_if_handle,
         _ip_address       address,
         _ip_address       netmask)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

address [in] — 设备接口的 IP 地址。

netmask [in] — 接口的网络掩码。

说明

函数 RTCS_if_bind()将 IP 地址和网络掩码绑定到与句柄 rtcs_if_handle 相关的设备接口。RTCS_if_add()返回参数 rtcs_if_handle。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind_BOOTP()
- RTCS_if_bind_DHCP()
- RTCS_if_bind_DHCP_flagged()
- RTCS_if_rebind_DHCP()

示例

请参见示例：设置 RTCS 。

7.129 RTCS_if_bind_BOOTP()

利用 BootP 获取 IP 地址并绑定到设备接口。

概要

```
uint32_t RTCS_if_bind_BOOTP(
        _rtcs_if_handle        rtcs_if_handle,
        BOOTP_DATA_STRUCT_PTR data_ptr)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄

data_ptr [in/out] — 指向 BootP 数据的指针

说明

该函数利用 BootP 来指定 IP 地址，确定要下载的启动文件，并确定从哪个服务器下载该文件。**RTCS_if_add()**返回参数 *rtcs_if_handle*。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()
- RTCS_if_bind_DHCP()
- RTCS_if_bind_IPCP()
- *BOOTP_DATA_STRUCT*

示例

```
BOOTP_DATA_STRUCT boot_data;

uint32_t boot_function(void)
{
  BOOTP_DATA_STRUCT  boot_data;
  _enet_handle       ehandle;
  _rtcs_if_handle    ihandle;
  uint32_t             error;

  error = ENET_initialize(0, enet_local, 0, &ehandle);

  if (error) return error;

  error = RTCS_create();
  if (error) return error;

  error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
  if (error) return error;

  memset(&boot_data, 0, sizeof(boot_data));
  error = RTCS_if_bind_BOOTP(ihandle, &boot_data);

  return error;
}
```

7.130 RTCS_if_bind_DHCP()

利用 DHCP 获取 IP 地址并绑定到设备接口。

概要

```
uint32_t RTCS_if_bind_DHCP(
        _rtcs_if_handle      rtcs_if_handle,
        DHCP_DATA_STRUCT_PTR callback_ptr,
        char               *optptr,
        uint32_t              optlen)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

callback_ptr [in] — 指向用于 DHCP 的回调函数的指针。

optptr [in] — 以下之一:

指向 DHCP 参数缓冲区的指针（见 RFC 2132）

NULL

optlen [in] — 由 *optptr* 指向的缓冲区中的字节数。

说明

函数 RTCS_if_bind_DHCP()利用 DHCP 获取 IP 地址并绑定到设备接口。RTCS_if_add()返回参数 rrtcs_if_handle。

该函数在 DHCP 完成初始化之前和绑定接口之后为阻塞。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()
- RTCS_if_bind_BOOTP()
- RTCS_if_bind_DHCP_flagged()
- RTCS_if_bind_DHCP_timed()
- RTCS_if_bind_IPCP()
- DHCP_DATA_STRUCT

示例

```
_enet_handle       ehandle;
_rtcs_if_handle    ihandle;
uint32_t           error;
uint32_t           optlen = 100; /* Use the size that you need for
                                   the number of params that you
                                   are using with DHCP */
uchar              option_array[100];
uchar *         optptr;
DHCP_DATA_STRUCT params;
uchar             parm_options[3] = {DHCPOPT_SERVERNAME,
                                    DHCPOPT_FILENAME,
                                    DHCPOPT_FINGER_SRV};

error = ENET_initialize(0, enet_local, 0, &ehandle);
if (error) {
 printf("\nFailed to initialize Ethernet driver: %s.",
           ENET_strerror(error));
```

```
    return;
  }

  error = RTCS_create();
  if (error != RTCS_OK) {
    printf("\nFailed to create RTCS, error = %x.", error);
    return;
  }

  error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
  if (error) {
    printf("\nFailed to add the interface, error = %x.", error);
    return;
  }

  /* You supply the following functions; if any is NULL, DHCP Client
     follows its default behavior. */
  params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
  params.BIND_FUNC = DHCPCLNT_test_bind_func;
  params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;

  optptr = option_array;
  /* Fill in the requested params: */
  /* Request a three-minute lease: */
  DHCP_option_int32(&optptr, &optlen, DHCPOPT_LEASE, 180);
  /* Request a TFTP Server, FILENAME, and Finger Server: */
  DHCP_option_variable(&optptr, &optlen, DHCPOPT_PARAMLIST,
                              parm_options, 3);

  error = RTCS_if_bind_DHCP(ihandle, &params, option_array,
                         optptr - option_array);
  if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
    return;
  }

  /* Use the network interface when it is bound. */
```

7.131 RTCS_if_bind_DHCP_flagged()

利用 DHCP 获取 IP 地址并绑定到设备接口，使用由 dhcp.h 中的标志所定义的参数。

概要

```
uint32_t RTCS_if_bind_DHCP_flagged(
        _rtcs_if_handle       rtcs_if_handle,
        DHCP_DATA_STRUCT_PTR params,
        char               *optptr,
        uint32_t             optlen)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

params [in] — 可选参数

params->CHOICE_FUNC

params->BIND_FUNC

params->REBIND_FUNC

params->UNBIND_FUNC

params->FAILURE_FUNC

params->FLAGS

optptr [in] — 以下之一:

指向 DHCP 参数缓冲区的指针（见 RFC 2132)。

NULL

optlen [in] — 由 *optptr* 指向的缓冲区中的字节数。

说明

函数 RTCS_if_bind_DHCP_flagged()利用 DHCP 获取 IP 地址并绑定到设备接口。TCPIP_PARM_IF_DHCP 结构在 dhcp_prv.h 中定义。FLAGS 在 dhcp.h 中定义。RTCS_if_add()返回参数 rtcs_if_handle。

欲使 DHCP 客户端不调试网络便接受所提供的 IP 地址，则不要在 params->FLAGS 中设置 DHCP_SEND_PROBE。

该函数在 DHCP 完成初始化之前和绑定接口之后为阻塞。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()
- RTCS_if_bind_BOOTP()
- RTCS_if_bind_IPCP()
- *DHCP_DATA_STRUCT*

示例

```
_enet_handle    ehandle;
_rtcs_if_handle ihandle;
uint32_t        error;
uint32_t        optlen = 100; /* Use the size that you need for
                                 the number of params that you
                                 are using with DHCP */
uchar           option_array[100];
uchar *     optptr;
DHCP_DATA_STRUCT params;
```

```
uchar            parm_options[3] = {DHCPOPT_SERVERNAME,
                                    DHCPOPT_FILENAME,
                                    DHCPOPT_FINGER_SRV};

error = ENET_initialize(0, enet_local, 0, &ehandle);
if (error) {
   printf("\nFailed to initialize Ethernet driver: %s.",
             ENET_strerror(error));
   return;
}

error = RTCS_create();
if (error != RTCS_OK) {
   printf("\nFailed to create RTCS, error = %x.", error);
   return;
}

error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
   printf("\nFailed to add the interface, error = %x.", error);
   return;
}

/* You supply the following functions; if any is NULL, DHCP Client
   follows its default behavior. */
params.FLAGS = 0;
params.FLAGS |= DHCP_SEND_INFORM_MESSAGE;
params.FLAGS |= DHCP_MAINTAIN_STATE_ON_INFINITE_LEASE;
params.FLAGS |= DHCP_SEND_PROBE;
params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
params.BIND_FUNC = DHCPCLNT_test_bind_func;
params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;

optptr = option_array;
/* Fill in the requested params: */
/* Request a three-minute lease: */
DHCP_option_int32(&optptr, &optlen, DHCPOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCPOPT_PARAMLIST,
                     parm_options, 3);

error = RTCS_if_bind_DHCP(ihandle, &params, option_array,
  if (error) {
    printf("\nDHCP boot failed, error = %x.", error);
    return;
}
/* Use the network interface when it is bound. */
```

7.132 RTCS_if_bind_DHCP_timed()

在超时前利用 DHCP 获取 IP 地址并绑定到设备接口。

概要

```
uint32_t RTCS_if_bind_DHCP_timed(
        _rtcs_if_handle       rtcs_if_handle,
        DHCP_DATA_STRUCT_PTR params,
        char                *optptr,
        uint32_t              optlen)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

params [in] — 可选参数

params->CHOICE_FUNC

params->BIND_FUNC

params->REBIND_FUNC

params->UNBIND_FUNC

params->FAILURE_FUNC

params->FLAGS

optptr [in] — 以下之一:

指向 DHCP 参数缓冲区的指针（见 RFC 2132）。

NULL。

optlen [in] — 由 *optptr* 指向的缓冲区中的字节数。

说明

函数 RTCS_if_bind_DHCP_timed()利用 DHCP 获取 IP 地址并绑定到设备接口。如果接口在超时限制前未通过 DHCP 绑定，则客户端停止尝试绑定并退出。RTCS_if_add()返回参数 rtcs_if_handle。

该函数在 DHCP 完成初始化之前和绑定接口之后为阻塞。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()
- RTCS_if_bind_BOOTP()
- RTCS_if_bind_IPCP()
- DHCP_DATA_STRUCT

示例

```
_enet_handle    ehandle;
_rtcs_if_handle ihandle;
uint32_t        error;
```

```
 uint32_t        optlen = 100; /* Use the size that you need for
                                the number of params that you
                                are using with DHCP */
 uchar           option_array[100];
 uchar *     optptr;
 DHCP_DATA_STRUCT params;
 uchar            parm_options[3] = {DHCPOPT_SERVERNAME,
                                    DHCPOPT_FILENAME,
                                    DHCPOPT_FINGER_SRV};
 uint32_t         timeout = 120; /* two minutes*/

 error = ENET_initialize(0, enet_local, 0, &ehandle);
 if (error) {
     printf("\nFailed to initialize Ethernet driver: %s.",
              ENET_strerror(error));
     return;
 }

 error = RTCS_create();
 if (error != RTCS_OK) {
   printf("\nFailed to create RTCS, error = %x.", error);
   return;
}

error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
if (error) {
    printf("\nFailed to add the interface, error = %x.", error);
    return;
}

/* You supply the following functions; if any is NULL, DHCP Client
   follows its default behavior. */
params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
params.BIND_FUNC = DHCPCLNT_test_bind_func;
params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;

optptr = option_array;
/* Fill in the requested params: */
/* Request a three-minute lease: */
DHCP_option_int32(&optptr, &optlen, DHCPOPT_LEASE, 180);
/* Request a TFTP Server, FILENAME, and Finger Server: */
DHCP_option_variable(&optptr, &optlen, DHCPOPT_PARAMLIST,
                     parm_options, 3);

error = RTCS_if_bind_DHCP_timed(ihandle, &params, option_array,
                        optptr - option_array, timeout);
if (error) {
  printf("\nDHCP boot failed, error = %x.", error);
  return;
}
/* Use the network interface if it successfully binds. Check
   after the timeout value to see if it did bind. */
```

7.133 RTCS_if_bind_IPCP()

将 IP 地址绑定到 PPP 设备接口。

概要

```
uint32_t RTCS_if_bind_IPCP(
        _rtcs_if_handle        rtcs_if_handle,
        IPCP_DATA_STRUCT_PTR   data_ptr)
```

参数

rtcs_if_handle [in] — 用于 PPP 设备的 RTCS 接口句柄。

data_ptr [in] — 指向 IPCP 数据的指针。

说明

只能通过函数 RTCS_if_bind_IPCP()将 IP 地址绑定到 PPP 设备接口。

函数开始与 PPP 接口协商 IPCP，这由 rtcs_if_handle 指定（返回 RTCS_if_add()）。函数立即返回；不会等待 IPCP 完成协商。IPCP_DATA_STRUCT 包含配置参数和一组应用回调函数当特定事件发生时，RTCS 会调用。更多详情，请参见第 8 章，"数据类型" 中的 IPCP_DATA_STRUCT 。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- PPP_init()
- RTCS_if_add()
- RTCS_if_bind()
- IPCP_DATA_STRUCT

示例

初始化 PPP 并绑定接口。

```
void boot_done(void *sem) {
  _lwsem_post(sem);
}

int32_t init_ppp(void)
{
  FILE_PTR         pppfile;
  _iopcb_handle    pppio;
  _ppp_handle      phandle;
  _rtcs_if_handle  ihandle;
  IPCP_DATA_STRUCT ipcp_data;
  LWSEM_STRUCT boot_sem;

  pppfile = fopen("ittya:", NULL);
  if (pppfile == NULL) return -1;
  pppio = _iopcb_ppphdlc_init(pppfile);
  if (pppio == NULL) return -1;
  error = PPP_initialize(pppio, &phandle);
  if (error) return error;
  _iopcb_open(pppio, PPP_lowerup, PPP_lowerdown, phandle);
  error = RTCS_if_add(phandle, RTCS_IF_PPP, &ihandle);
  if (error) return error;

  _lwsem_create(&boot_sem, 0);
```

```
    memset(&ipcp_data, 0, sizeof(ipcp_data));
    ipcp_data.IP_UP                = boot_done;
    ipcp_data.IP_DOWN              = NULL;
    ipcp_data.IP_PARAM             = &boot_sem;
    ipcp_data.ACCEPT_LOCAL_ADDR    = FALSE;
    ipcp_data.ACCEPT_REMOTE_ADDR   = FALSE;
    ipcp_data.LOCAL_ADDR           = PPP_LOCADDR;
    ipcp_data.REMOTE_ADDR          = PPP_PEERADDR;
    ipcp_data.DEFAULT_NETMASK      = TRUE;
    ipcp_data.DEFAULT_ROUTE        = TRUE;

    error = RTCS_if_bind_IPCP(ihandle, &ipcp_data);
    if (error) return error;

    _lwsem_wait(&boot_sem);
    printf("IPCP is up\n");
    return 0;
}
```

7.134 RTCS_if_rebind_DHCP()

将之前使用的 IP 地址绑定到设备接口。

概要

```
uint32_t RTCS_if_rebind_DHCP(
        _rtcs_if_handle         rtcs_if_handle,
        _ip_address             address,
        _ip_address             netmask,
        uint32_t                  lease,
        _ip_address               server,
        DHCP_DATA_STRUCT_PTR      params,
        unsigned char                 *optptr,
        uint32_t                    optlen)
```

参数

handle [in] — RTCS 接口句柄。

address [in] — 用于接口的 IP 地址。

netmask [in] — 用于接口的网络或子网掩码的 IP 地址。

lease [in] — 租约的持续时间（以秒为单位）。

server [in] — DHCP 服务器的 IP 地址。

params — 可选参数

params->CHOICE_FUNC

params->BIND_FUNC

params->REBIND_FUNC

params->UNBIND_FUNC

params->FAILURE_FUNC

params->FLAGS

optptr [in] — 以下之一:

指向 DHCP 选项缓冲区的指针（见 RFC 2132）。

NULL。

optlen [in] — 由 *optptr* 指向的缓冲区中的字节数。

说明

函数 RTCS_if_rebind_DHCP()利用 DHCP 获取 IP 地址并绑定到设备接口。RTCS_if_add()返回参数 rtcs_if_handle。

该函数在 DHCP 完成初始化之前和绑定接口之后为阻塞。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()
- RTCS_if_bind_BOOTP()
- RTCS_if_bind_DHCP_flagged()
- RTCS_if_bind_DHCP_timed()
- RTCS_if_bind_IPCP()
- DHCP_DATA_STRUCT

示例

```
Example
    _enet_handle    ehandle;
    _rtcs_if_handle ihandle;
    uint32_t        error;
    uint32_t        optlen = 100; /* Make large enough for the number
                                   of your DHCP options */
    uchar           option_array[100];
    uchar *      optptr;
    DHCP_DATA_STRUCT params;
    uchar            parm_options[3] = {DHCPOPT_SERVERNAME,
                                        DHCPOPT_FILENAME,
                                        DHCPOPT_FINGER_SRV};
    in_addr          rebind_address, rebind_mask, rebind_server;
    uint32_t         lease = 28800; /* 8 Hours, in seconds */
    error = ENET_initialize(0, enet_local, 0, &ehandle);
```

```
    if (error) {
       printf("\nFailed to initialize Ethernet driver: %s.",
                ENET_strerror(error));
       return;
    }
    error = RTCS_create();
    if (error != RTCS_OK) {
       printf("\nFailed to create RTCS, error = %x.", error);
       return;
    }
    error = RTCS_if_add(ehandle, RTCS_IF_ENET, &ihandle);
    if (error) {
       printf("\nFailed to add the interface, error = %x.", error);
       return;
    }
    /* You supply the following functions; if any is NULL, DHCP Client
       follows its default behavior. */
    params.CHOICE_FUNC = DHCPCLNT_test_choice_func;
    params.BIND_FUNC = DHCPCLNT_test_bind_func;
    params.UNBIND_FUNC = DHCPCLNT_test_unbind_func;
    optptr = option_array;
    /* Fill in the requested options: */
    /* Request a three-minute lease: */
    DHCP_option_int32(&optptr, &optlen, DHCPOPT_LEASE, 180);
    /* Request a TFTP Server, FILENAME, and Finger Server: */
    DHCP_option_variable(&optptr, &optlen, DHCPOPT_PARAMLIST,
                         parm_options, 3);
    error = inet_aton ("192.168.1.100", &rebind_address);
    error |= inet_aton ("255.255.255.0", &rebind_mask);
    error |= inet_aton ("192.168.1.2", &rebind_server);
    if (error) {
       printf("\nFailed to convert IP addresses from dotted decimal, error = %x.", error);
       return;
    }
    error = RTCS_if_rebind_DHCP(ihandle,
                                rebind_address,
                                rebind_mask,
                                lease,
                                rebind_server,
                                &params,
                                option_array,
                                optptr - option_array);
    if (error) {
       printf("\nDHCP boot failed, error = %x.", error);
       return;
    }
```

7.135 RTCS_if_remove()

从 RTCS 删除设备接口。

概要

```
uint32_t RTCS_if_remove(
        _rtcs_if_handle rtcs_if_handle)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

说明

函数 RTCS_if_remove()删除与 rtcs_if_handle 相关的设备接口（返回 RTCS_if_add()）。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_rebind_DHCP()

7.136 RTCS_if_get_link_status ()

返回接口的实际链接状态。

概要

```
bool RTCS_if_get_link_status(_rtcs_if_handle ihandle)
```

参数

ihandle [in] — 接口句柄

说明

该函数返回给定接口的实际链接状态。

返回值

- TRUE，如果接口链接为有效。
- FALSE，如果接口链接为无效。

另请参见

- RTCS_if_get_handle ()

示例

```
/* Print link status.*/
{
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
    bool            link;;

   link = RTCS_if_get_link_status(ihandle);
```

```
        printf ("Link : %s\n", link ? "on" : "off");
    }
```

7.137 RTCS_if_unbind()

取消绑定设备接口的 IP 地址。

概要

```
uint32_t RTCS_if_unbind(
        _rtcs_if_handle    rtcs_if_handle,
        _ip_address        address)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

address [in] — 要取消绑定的 IP 地址。

说明

函数 RTCS_if_unbind()将取消绑定与 rtcs_if_handle 相关的设备接口的 IP 地址。RTCS_if_add()返回参数 rtcs_if_handle。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS_if_add()
- RTCS_if_bind()
- RTCS_if_bind_BOOTP()
- RTCS_if_bind_DHCP()
- RTCS_if_bind_IPCP()
- RTCS_if_rebind_DHCP()

7.138 RTCS_if_get_dns_addr ()

返回给定设备接口的 DNS 地址列表中的第 n 个 DNS IPv4 地址。

概要

```
bool RTCS_if_get_dns_addr(_rtcs_if_handle ihandle, uint32_t n, _ip_address *dns_addr)
```

参数

ihandle [in] — RTCS 接口句柄

n [in] — DNS IPv4 地址索引（从 0 开始）

dns_addr [out] — 指向 DNS IPv4 地址的指针

说明

该函数用于检索给定设备接口的 DNS IPv4 地址。

返回值

- TRUE（成功，填充了 dns_addr）
- FALSE（失败，不存在第 *n* 个 DNS 地址）

另请参见

- RTCS6_if_add_dns_addr ()
- RTCS6_if_del_dns_addr ()

示例

```
/* Print all DNS IPv4 addresses.*/
{
   _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
   char            addr_str[RTCS_IP4_ADDR_STR_SIZE];
   int             i;
   _ip_address     dns_addr;

   for(i=0; (RTCS_if_get_dns_addr(ihandle, i, &dns_addr) == TRUE); i++)
   {

      printf ("[%d]: %s\n", i + 1, inet_ntop(AF_INET, &dns_addr, addr_str,
sizeof(addr_str)));
      }
   }
```

7.139 RTCS_if_add_dns_addr ()

注册设备接口的 DNS IPv4 地址。

概要

```
uint32_t RTCS_if_add_dns_addr(_rtcs_if_handle ihandle, _ip_address dns_addr)
```

参数

ihandle [in] — RTCS 接口句柄

dns_addr [in] — 要添加的 DNS IPv4 地址

说明

该函数将 DNS IPv4 地址添加到给定设备接口指定的列表。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- RTCS6_if_get_dns_addr ()
- RTCS6_if_del_dns_addr ()

示例

```
   /* Register DNS IPv4 address with the device interfae.*/
   {

      _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
      char            *addr_str = "8.8.8.8";
      _ip_address        dns_addr;

      if(inet_pton(AF_INET, addr_str, &dns_addr, sizeof(dns_addr)) == RTCS_OK)
   {

      if(RTCS_if_add_dns_addr(ihandle, dns_addr) == RTCS_OK)
      {
         printf("Adding DNS address is successful.\n");
      }
      else
      {
         printf("Adding DNS address is failed.\n");
      }
   }
}
```

7.140 RTCS_if_del_dns_addr ()

从设备接口取消注册 DNS IPv4 地址。

概要

```
uint32_t RTCS_if_del_dns_addr(_rtcs_if_handle ihandle, _ip_address dns_addr)
```

参数

ihandle [in] — RTCS 接口句柄

dns_addr [in] — 要删除的 DNS IPv4 地址

说明

该函数从给定设备接口指定的列表中删除 DNS IPv4 地址。

返回值

- *RTCS_OK*（成功）
- 错误代码（失败）

另请参见

- RTCS6_if_get_dns_addr ()
- RTCS6_if_add_dns_addr ()

示例

```
/* Unregister DNS IPv4 address from the device interface.*/
{
   _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
   char             *addr_str = "8.8.8.8";
   _ip_address      dns_addr;

   if(inet_pton(AF_INET, addr_str, &dns_addr, sizeof(dns_addr)) == RTCS_OK)
   {

     if(RTCS_if_del_dns_addr(ihandle, dns_addr) == RTCS_OK)
     {

       printf("Deleting DNS address is successful.\n");
     }
     else
     {

       printf("Deleting DNS address is failed.\n");
     }
   }
}
```

7.141 RTCS_ping()

向 IP 地址发送 ICMP 回送检验（echo）请求消息并等待应答。

概要

```
uint32_t RTCS_ping(PING_PARAM_STRUCT *params)
```

参数

params [in] — 指向 PING_PARAM_STRUCT 参数结构的指针，由 PING 函数使用。不应为 NULL。

说明

函数 RTCS_ping()为 RTCS 实现 Ping。它向特定的 IPv4 或 IPv6 地址发送 ICMPv4 或 ICMPv6 回送检验请求消息并等待应答。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- PING_PARAM_STRUCT

示例

```
    /* Send ICMPv4 echo request to the IPv4 192.168.0.5 address.*/
    {
       uint32_t           error;
       PING_PARAM_STRUCT ping_params;

       /* Set ping parameters.*/
       _mem_zero(&ping_params, sizeof(ping_params)); /* Zero input parameters.*/
       ping_params.addr.sa_family = AF_INET; /* Set IPv4 addr. family */
       /* IPv4 192.168.0.5 address.*/
      ((sockaddr_in *)(&ping_params.addr))->sin_addr.s_addr =  IPADDR(192,168,0,5);
      /* Wait interval in milliseconds */
      ping_params.timeout = 1000;

      /* Send PING - ICMP request.
      * It will block the application while await ICMP echo reply.*/
      error = RTCS_ping(&ping_params);

     if (error)
     {

        if (error == RTCSERR_ICMP_ECHO_TIMEOUT)
            printf("Request timed out\n");

        else
            printf("Error 0x%04lX \n", error);
    }
    else
    {
        if(ping_params.round_trip_time < 1)
            printf("Reply time<1ms\n");
        else
           printf("Reply time=%ldms\n", ping_params.round_trip_time);
    }
}
```

7.142 RTCS_request_DHCP_inform()

请求 DHCP 信息性消息。

概要

```
uint32_t RTCS_request_DHCP_inform(
          _rtcs_if_handle           handle,
          unsigned char                 *optptr,
          uint32_t                  optlen,
          _ip_address              client_addr,
          _ip_address              server_addr,
          void                     (_CODE_PTR_ inform_func)(uchar _PTR_,
                             uint32_t, _rtcs_if_handle))
```

参数

handle [in] — RTCS 接口句柄。

optptr [in] — 以下之一:

指向 DHCP 选项缓冲区的指针 (见 RFC 2132)。

NULL。

optlen [in] — 由 *optptr* 指向的缓冲区中的字节数。

client_addr [in] — 应用绑定的 IP 地址。

server_addr [in] — 用于所需信息的服务器 IP 地址。

inform_func — 当 DHCP 完成时所要调用的函数。

说明

函数 RTCS_request_DHCP_inform()请求关于服务器的信息消息。

返回值

- 服务器 DHCP 信息 (成功)
- 错误代码 (失败)

7.143 RTCS_selectall()

建议将 select()函数用于新应用。

如果使能了选项 RTCSCFG_SOCKET_OWNERSHIP, 然后该函数会等待调用任务所有的任意套接字上的活动。否则, 它将等待任意套接字上的活动。

概要

```
uint32_t RTCS_selectall(
        uint32_t timeout)
```

参数

timeout [in] — 以下之一:

等待活动的最大毫秒数。

0 （无限期等待）。

-1 （不阻塞）。

说明

如果 timeout 不为-1，该函数将阻塞直到在调用任务所有的任意套接字上检测到活动。活动包含以下内容。

套接字	接收
取消绑定数据报	数据报
监听数据流	连接请求
已连接数据流	由远程端点发起的数据或关断请求。

返回值

- 套接字句柄 （检测到活动）
- 0 （timeout 超时）
- RTCS_SOCKET_ERROR （错误）

另请参见

- RTCS_attachsock()
- RTCS_detachsock()
- RTCS_selectset()

示例

TCP 端口号 7 上的回送检验数据。

```
 int32_t                  servsock;
 int32_t                  connsock;
 int32_t                  status;
 SOCKET_ADDRESS_STRUCT    addrpeer;
 uint16_t                  addrlen;
 char                     buf[500];
 int32_t                   count;
 uint32_t                  error

/* create a stream socket and bind it to port 7: */
error = listen(servsock, 0);
if (error != RTCS_OK) {
    printf("\nlisten() failed, status = %d", error);
    return;
}
for (;;) {
    connsock = RTCS_selectall(0);
```

```
        if (connsock == RTCS_SOCKET_ERROR) {
          printf("\nRTCS_selectall() failed!");

    } else if (connsock == servsock) {
      status = accept(servsock, &addrpeer, &addrlen);
      if (status == RTCS_SOCKET_ERROR)
       printf("\naccept() failed!");
       service_accept_error();

    } else {
      count = recv(connsock, buf, 500, 0);
      if (count <= 0)
        shutdown(connsock, FLAG_CLOSE_TX);
      else
        send(connsock, buf, count, 0);
    }
  }
```

7.144 RTCS_selectset()

建议将 select()函数用于新应用。

等待套接字集中的任意套接字上的活动。

概要

```
uint32_t RTCS_selectset(
      void *sockset,
      uint32_t count,
      uint32_t timeout)
```

参数

sockset [in] — 指向套接字数组的指针。

count [in] — 数组中的套接字数量。

timeout [in] — 以下之一:

等待活动的最大毫秒数。

0 (无限期等待)。

-1 (不阻塞)。

说明

如果 timeout 不为-1，函数将阻塞直到在套接字集中的至少一个套接字上检测到活动。对于活动包含内容的描述，请参见 RTCS_selectall() 。

返回值

- 套接字句柄 (检测到活动)

- 0（timeout 超时）
- RTCS_SOCKET_ERROR（错误）

另请参见

- RTCS_selectall()

示例

在端口 2010、2011 和 2012 上接收到回送检验 UDP 数据。

```
int32_t socklist[3];
sockaddr_in local_sin;
uint32_t result;
...
memset((char *) &local_sin, 0, sizeof(local_sin));
local_sin.sin_family = AF_INET;
local_sin.sin_addr.s_addr = INADDR_ANY;
local_sin.sin_port = 2010;
socklist[0] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[0], (struct sockaddr *)&local_sin, sizeof (sockaddr_in));
local_sin.sin_port = 2011;
socklist[1] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[1], (struct sockaddr *)&local_sin, sizeof (sockaddr_in));
local_sin.sin_port = 2012;
socklist[2] = socket(AF_INET, SOCK_DGRAM, 0);
result = bind(socklist[2], (struct sockaddr *)&local_sin, sizeof (sockaddr_in));
while (TRUE) {
   sock = RTCS_selectset(socklist, 3, 0);

   rlen = sizeof(raddr);
   length = recvfrom(sock, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&raddr, &rlen);
   sendto(sock, buffer, length, 0, (struct sockaddr *)&raddr, rlen);
}
```

7.145 RTCSLOG_disable()

禁止 RTCS 记录。

概要

```
    void RTCSLOG_disable(
            uint32_t logtype)
```

参数

logtype [in] — 停止记录一个或多个类条目。

说明

该函数禁止在 MQX 内核日志中记录 RTCS 事件。logtype 为按位，或以下之一:

- RTCS_LOGCTRL_FUNCTION — 记录所有套接字 API 调用。

- RTCS_LOGCTRL_PCB — 记录数据包生成和解析。
- 或者，logtype 可为 RTCS_LOGCTRL_ALL 禁止所有类的日志条目。

另请参见

RTCSLOG_enable()

示例

请参见 RTCSLOG_enable() 。

7.146 RTCSLOG_enable()

使能 RTCS 记录。

概要

```
void RTCSLOG_enable(
          uint32_t   logtype)
```

参数

logtype [in] — 开始记录一个或多个类条目。

说明

该函数使能在 MQX 内核日志中记录 RTCS 事件。logtype 为按位，或以下之一:

- RTCS_LOGCTRL_FUNCTION — 记录所有套接字 API 调用。
- RTCS_LOGCTRL_PCB — 记录数据包产生和解析。
- 或者，logtype 可为 RTCS_LOGCTRL_ALL 来使能所有类的日志条目。

RTCS 日志条目写入内核日志。因此，必须在使能 RTCS 记录前创建内核日志。

此外，套接字 API 日志条目属于内核中的内核日志函数组。要记录套接字 API 调用，该组必须通过 MQX 函数_klog_control()使能。

另请参见

- RTCSLOG_disable()
- 《*MQX RTOS* 参考手册》中的_klog_create()
- 《*MQX RTOS* 参考手册》中的_klog_control()

示例

创建内核日志。

```
_klog_create(16384, 0);
/* Tell MQX to log RTCS functions */
_klog_control(KLOG_ENABLED | KLOG_FUNCTIONS_ENABLED |
 RTCSLOG_FNBASE, TRUE);
/* Tell RTCS to start logging */
RTCSLOG_enable(RTCS_LOGCTRL_ALL);

/* ... */

/* Tell RTCS to stop logging */
RTCSLOG_disable(RTCS_LOGCTRL_ALL);
```

7.147 RTCS6_if_bind_addr()

将 IPv6 地址绑定到设备接口。

概要

```
uint32_t RTCS6_if_bind_addr (_rtcs_if_handle rtcs_if_handle, in6_addr *address,
rtcs6_if_addr_type address_type, uint32_t addr_lifetime)
```

参数

rtcs_if_handle [in] — RTCS 接口句柄。

address [in] — 设备接口的 IPv6 地址。

address_type [in] — IPv6 地址类型。它定义了向接口指定 IPv6 地址的方法:

- IP6_ADDR_TYPE_MANUAL — 地址参数值定义了要绑定到接口的整个 IPv6 地址。
- IP6_ADDR_TYPE_AUTOCONFIGURABLE — *address* 参数值定义了绑定 IPv6 地址的前 64 位。IPv6 地址的后 64 位由接口标识符覆盖。以以太网接口为例，接口标识符形式为 48 位 MAC 地址（根据[RFC2464]）。

addr_lifetime [in] — IPv6 地址的有效生存期（以秒为单位）。0xFFFFFFFF 值意味着无限生存期。

说明

函数 RTCS6_if_bind_addr()将 IPv6 地址绑定到与句柄 rtcs_if_handle 相关的设备接口。RTCS_if_add()返回参数 rtcs_if_handle。

一个接口可能具有多个绑定的 IPv6 地址。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS6_if_unbind_addr()
- ip6_if_addr_type

示例

```
    /* Bind 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
    {
      /* Before, interface was initialized by ipcfg_init_device().*/
      _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);
      /* Bind 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
      in6_addr        address = IN6ADDR(0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,
                                        0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf);
      uint32_t        error;

      if(ihandle)
      {

        error = RTCS6_if_bind_addr(ihandle, &address, IP6_ADDR_TYPE_MANUAL,
IP6_ADDR_LIFETIME_INFINITE, IP6_ADDR_LIFETIME_INFINITE);
        if (error == RTCS_OK)
            printf("The interface is bound.\n");
        else
           printf("Failed to bind interface, error = %x\n", error);
    }
    else
        printf("Not initialized by ipcfg_init_device().\n");
}
```

7.148 RTCS6_if_unbind_addr()

从设备接口解绑 IPv6 地址。

概要

```
uint32_t RTCS6_if_unbind_addr (_rtcs_if_handle  rtcs_if_handle, in6_addr *address)
```

参数

- *rtcs_if_handle [in]* — RTCS 接口句柄。
- *address [in]* — 要解绑的 IPv6 地址。

说明

函数 RTCS6_if_unbind_addr()将解绑 rtcs_if_handle 相关的设备接口的 IPv6 地址。RTCS_if_add()返回参数 rtcs_if_handle。

返回值

- RTCS_OK（成功）
- 错误代码（失败）

另请参见

- RTCS6_if_bind_addr()

示例

```
/* Unbind 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
{
    uint32_t        error;
    /* Before, interface was initialized by ipcfg_init_device().*/
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);
    /* 1:203:405:607:809:a0b:c0d:e0f IPv6 address.*/
    in6_addr         address = IN6ADDR(0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,
                                        0x8,0x9,0xa,0xb,0xc,0xd,0xe,0xf);
    if(ihandle)
    {
      error = RTCS6_if_bind_addr(ihandle, &address, IP6_ADDR_TYPE_MANUAL);
      if (error == RTCS_OK)
      {
          printf("The interface is bound.\n");

          error = RTCS6_if_unbind_addr (ihandle, &address);

          if (error == RTCS_OK)
              printf("The interface is unbound.\n");
              printf("Failed to unbind interface, error = %x\n", error);
      }
      else
         printf("Failed to bind interface, error = %x\n", error);
    }
   else
      printf("Not initialized by ipcfg_init_device().\n");
}
```

7.149 RTCS6_if_get_scope_id()

返回分配给设备接口的范围 ID。

概要

```
uint32_t RTCS6_if_get_scope_id (_rtcs_if_handle rtcs_if_handle)
```

参数

- *rtcs_if_handle [in]* — RTCS 接口句柄。

说明

该函数返回分配给与 rtcs_if_handle 相关的设备接口的范围 ID（接口标识符）。范围 ID 用于指示网络接口使用的发送/接收途径。

返回值

- 范围 ID（成功）
- 0（失败）

另请参见

- RTCS6_if_bind_addr()

示例

```
/* Get Scope ID assigned to the interface.*/
{
   uint32_t scope_id;
   /* Before, interface was initialized by ipcfg_init_device().*/
   _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);

   if(ihandle)
   {

     scope_id = RTCS6_if_get_scope_id(ihandle);
     if(scope_id == 0)
        printf("Scope ID is not assigned to the interface.\n");
      else
        printf("Scope ID = %x\n", scope_id);
    }
    else
        printf("Not initialized by ipcfg_init_device().\n");
}
```

7.150 RTCS6_if_get_prefix_list_entry()

返回设备接口的 IPv6 前缀列表内容。

概要

```
bool RTCS6_if_get_prefix_list_entry(_rtcs_if_handle ihandle, uint32_t n,
RTCS6_IF_PREFIX_LIST_ENTRY_PTR prefix_list_entry)
```

参数

ihandle [in] — RTCS 接口句柄

n [in] — IPv6 前缀索引（从 0 开始）

prefix_list_entry [in/out] — 指向 IPv6 前缀列表入口的指针

说明

该函数可用于检索给定设备接口的 IPv6 前缀列表内容。

该函数主要用于测试或获取信息。

返回值

- *TRUE*（成功，填充了 *prefix_list_entry*）
- *FALSE*（失败，不存在第 *n* 个前缀）

另请参见

- RTCS6_if_get_neighbor_cache_entry()

示例

```
/* Print IPv6 Prefix List. */
{
    _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
    char            addr_str[RTCS_IP6_ADDR_STR_SIZE];
    int             i;
    RTCS6_IF_PREFIX_LIST_ENTRY     prefix_list_entry;

    printf("\nIPv6 Prefix List:\n");
    for(i=0; RTCS6_if_get_prefix_list_entry(ihandle, i, &prefix_list_entry) == TRUE; i++)
    {
        printf(" [%d] %s/%d\n", i,
            inet_ntop(AF_INET6, &prefix_list_entry.prefix, addr_str,
                sizeof(addr_str)), prefix_list_entry.prefix_length);
    }
 }
```

7.151 RTCS6_if_get_neighbor_cache_entry()

返回设备接口的 IPv6 相邻缓存内容。

概要

```
bool RTCS6_if_get_neighbor_cache_entry(_rtcs_if_handle ihandle, uint32_t n,
RTCS6_IF_NEIGHBOR_CACHE_ENTRY_PTR neighbor_cache_entry)
```

参数

ihandle [in] — RTCS 接口句柄

n [in] — IPv6 前缀索引（从 0 开始）

neighbor_cache_entry [in/out] — 指向 IPv6 相邻缓存入口的指针

说明

该函数可用于检索给定设备接口的 IPv6 相邻缓存内容。

该函数主要用于测试或获取信息。

返回值

- *TRUE*（成功，填充了 *neighbor_cache_entry*）
- *FALSE*（失败，不存在第 *n* 个相邻缓存入口）

另请参见

- RTCS6_IF_NEIGHBOR_CACHE_ENTRY

示例

```
    /* Print IPv6 Prefix List. */
    {
       _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
       char            addr_str[RTCS_IP6_ADDR_STR_SIZE];
       int             i;
       RTCS6_IF_NEIGHBOR_CACHE_ENTRY neighbor_cache_entry;

       printf("\nIPv6 Neighbor Cache:\n");
       for(i=0; RTCS6_if_get_neighbor_cache_entry(ihandle, i, &neighbor_cache_entry) == TRUE;
i++)
      {
           printf("  [%d] %s = %02x:%02x:%02x:%02x:%02x:%02x (%s) \n", i,
                    inet_ntop(AF_INET6, &neighbor_cache_entry.ip_addr,
                    addr_str, sizeof(addr_str)),
                    neighbor_cache_entry.ll_addr[0],
                    neighbor_cache_entry.ll_addr[1],
                    neighbor_cache_entry.ll_addr[2],
                    neighbor_cache_entry.ll_addr[3],
                    neighbor_cache_entry.ll_addr[4],
                    neighbor_cache_entry.ll_addr[5],
                   (neighbor_cache_entry.is_router == TRUE)? "router" : "host" );
        }
}
```

7.152 RTCS6_if_get_addr()

返回绑定至设备接口的 IPv6 地址信息。

概要

```
uint32_t RTCS6_if_get_addr(_rtcs_if_handle ihandle, uint32_t n, RTCS6_IF_ADDR_INFO
*addr_info)
```

参数

- *rtcs_if_handle [in]* — RTCS 接口句柄。
- *n [in]* — 要检索的 IPv6 地址序列号（从 0 开始）。
- *addr_info [in/out]* — 指向 IPv6 地址信息（IPv6 地址、地址状态和类型）的指针。

说明

该函数返回绑定至给定设备接口的 IPv6 地址信息。

一个接口可能具有多个绑定的 IPv6 地址。

返回值

- RTCS_OK（成功，填充了 addr_info）
- RTCS_ERROR（失败，不存在第 n 个地址）

另请参见

- RTCS6_if_bind_addr()
- RTCS6_IF_ADDR_INFO

示例

```
    /* Print all bound IPv6 addresses.*/
    {
       /* Before, interface was initialized by ipcfg_init_device().*/
       _rtcs_if_handle ihandle = ipcfg_get_ihandle(0);
       char prn_addr6[RTCS_IP6_ADDR_STR_SIZE];

       if(ihandle)
       {

          RTCS6_IF_ADDR_INFO addr_info;
          int                n=0;

         /* Print all bound IPv6 addresses.*/
         while(RTCS6_if_get_addr(ihandle, n, &addr_info) == RTCS_OK)
         {

           /* Convert IPv6 address to string presentation and print it.*/
           if(inet_ntop(AF_INET6, &addr_info.ip_addr, prn_addr6,  sizeof(prn_addr6)))
           {
             printf("IP6[%d] : %s\n", n, prn_addr6);
          }
          n++;
       }
    }
    else
       printf("Not initialized by ipcfg_init_device().\n");
}
```

7.153 RTCS6_if_get_dns_addr ()

返回设备接口的已注册 DNS 列表中的第 n 个 DNS IPv6 地址。

概要

```
bool RTCS6_if_get_dns_addr(_rtcs_if_handle ihandle, uint32_t n, in6_addr *dns_addr)
```

参数

- *ihandle [in]* — RTCS 接口句柄
- *n [in]* — DNS IPv6 地址索引（从 0 开始）
- *dns_addr [out]* — 指向 DNS IPv6 地址的指针

说明

该函数可用于检索所有已注册 (手动或通过 IPv6 路由器发现流程) 给定设备接口的 DNS IPv6 地址。

返回值

- RTCS_OK（成功，填充了 addr_info）
- RTCS_ERROR（失败，不存在第 n 个地址）

另请参见

- RTCS6_if_add_dns_addr ()
- RTCS6_if_del_dns_addr ()

示例

```
/* Print all DNS IPv6 addresses.*/
   {
        _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
        char            addr_str[RTCS_IP6_ADDR_STR_SIZE];
        int             i;
        in6_addr        dns6_addr;

        for(i=0; (RTCS6_if_get_dns_addr(ihandle, i, &dns6_addr) == TRUE); i++)
        {

           printf ("%d: %s\n", i + 1, inet_ntop(AF_INET6, &dns6_addr, addr_str,
sizeof(addr_str)));
       }
         }
```

7.154 RTCS6_if_add_dns_addr ()

注册设备接口的 DNS IPv6 地址。

概要

```
uint32_t RTCS6_if_add_dns_addr(_rtcs_if_handle ihandle, in6_addr *dns_addr)
```

参数

- *ihandle [in]* — RTCS 接口句柄。
- *dns_addr [out]* — 指向 DNS IPv6 地址的指针。

说明

该函数将 DNS IPv6 地址添加到给定设备接口指定的列表。

返回值

- RTCS_OK（成功，填充了 addr_info）
- RTCS_ERROR（失败，不存在第 n 个地址）

另请参见

- RTCS6_if_get_dns_addr ()
- RTCS6_if_del_dns_addr ()

示例

```
/* Register DNS IPv6 address with the device interfae.*/
    {
        _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
        char *addr_str = "2001:470:1234:567:4c39:64fa:1caa:44c8";
        in6_addr dns6_addr;

        if(inet_pton(AF_INET6, addr_str, &dns6_addr, sizeof(dns6_addr)) == RTCS_OK)
       {
          if(RTCS6_if_add_dns_addr(ihandle, &dns6_addr) == RTCS_OK)
          {
             printf("Adding DNS address is successful.\n");
          }
          else
          {
             printf("Adding DNS address is failed.\n");
          }
       }
         }
```

7.155 RTCS6_if_del_dns_addr ()

从设备接口取消注册 DNS IPv6 地址。

概要

```
uint32_t RTCS6_if_del_dns_addr(_rtcs_if_handle ihandle, in6_addr *dns_addr)
```

参数

- *ihandle [in]* — RTCS 接口句柄。
- *dns_addr [in]* — 要删除的 DNS IPv6 地址。

说明

该函数从给定设备接口指定的列表中删除 DNS IPv6 地址。

返回值

- *RTCS_OK* （成功）
- 错误代码 （失败）

另请参见

- RTCS6_if_get_dns_addr ()
- RTCS6_if_del_dns_addr ()

示例

```
    /* Unregister DNS IPv6 address from the device interface.*/
        {
           _rtcs_if_handle ihandle = ipcfg_get_ihandle(BSP_DEFAULT_ENET_DEVICE);
char                  *addr_str = "2001:470:1234:567:4c39:64fa:1caa:44c8";
in6_addr              dns6_addr;
if(inet_pton(AF_INET6, addr_str, &dns6_addr, sizeof(dns6_addr)) == RTCS_OK)
{
   if(RTCS6_if_del_dns_addr(ihandle, &dns6_addr) == RTCS_OK)
   {
      printf("Deleting DNS address is successful.\n");
   }
   else
   {
      printf("Deleting DNS address is failed.\n");
   }
  }
}
```

7.156 RTCS6_if_is_disabled()

检测接口是否禁止了 IPv6。

概要

```
_ip_address RTCS_if_get_addr(_rtcs_if_handle ihandle)
```

参数

- *ihandle [in]* — RTCS 接口句柄

说明

该函数用于检测接口是否禁止了 IPv6。如果发生以下情况，可以禁止 IPv6 操作：

- 由于以基于硬件地址的接口标识符为本地链路地址形式的地址导致重复地址检测失败，该地址应为唯一指定（例如，EUI-64 用于以太网接口）。
- RTCSCFG_ENABLE_IP6 设为 0。
- 如果使用了 IPv6 评估库，则过期一个操作超时。

返回值

- TRUE，如果禁止了 IPv6 操作。
- FALSE，如果使能了 IPv6 操作。

7.157 select()

该功能查询套接字描述符并检查是否有等待的连接/数据/关闭请求。如果对于任意给定套接字没有正在等待的请求，该函数可以阻塞直到 RTCS 提示有请求。

概要

```
int32_t select(int32_t nfds,
                    rtcs_fd_set *restrict readfds,
                    rtcs_fd_set *restrict writefds,
                    rtcs_fd_set *restrict exceptfds,
                    uint32_t timeout_ms);
```

参数

- int32_t nfds [IN]

  开头的 nfds 套接字句柄在各自集中检查，意味着检验描述符集中的套接字 0 至 nfds-1。

- rtcs_fd_set * readfds [IN/OUT]

  IN - 指向 SOCKET_STRUCT 的指针数组，用于检验接收活动

  OUT - Readfds 包含检测到活动的套接字句柄

- rtcs_fd_set * writefds [IN/OUT]

  IN - 指向 SOCKET_STRUCT 的指针数组，用于检验发送活动

  OUT - Writefds 包含检测到活动的套接字句柄

- rtcs_fd_set * exceptfds [IN/OUT]

  IN - 指向 SOCKET_STRUCT 的指针数组，用于检验 RTCS 异常信号。

OUT - exceptfds 包含检测到 RTCS 异常信号的套接字句柄。

uint32_t timeout_ms [IN]

- 如果 timeout_ms 为 0，select()可能永久阻塞。如果 timeout_ms 为-1，select()仅查询套接字描述符并在确定实际状态时返回。timeout_ms 的其他值确定了 select()函数阻塞的最大时间（以毫秒为单位）。

说明

由于限制关键字，rtcs_fd_sets 不能重叠。Readfds、writefds 和 exceptfds 在存储器中不能指向相同的 rtcs_fd_set 结构。

如果没有相关描述符，任意 rtcs_fd_set 指针可成为空指针。select()函数修改 rtcs_fd_set 数组的内容。

如果数据流套接字位于输入 readfds 且满足以下条件之一，则会通过 readfds 返回:

- 请求连接且套接字正在监听
- 请求关闭
- 数据可供读取

如果数据报套接字位于输入 readfds 且满足以下条件之一，则会通过 readfds 返回:

- 数据可供读取

如果数据流套接字位于输入 writefds 且满足以下条件之一, 则会通过 writefds 返回:

- 发送缓冲区为空
- 所有发送数据被远程对等端应答且发送缓冲区为空

如果数据报套接字位于输入 writefds，它将总是通过 writefds 返回。这是由于当前 RTCS 的设置。应用应仍然从 sendto()/send()函数检查返回值。

建议将 Select()函数用于新应用。提供建立时间配置参数 RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT 用于依靠 RTCS_selectset()和/或 RTCS_selectall()函数的工程的向下兼容性。

第三个 rtcs_fd_set 参数 exceptfds 在 RTCS 中有特殊用途。它可通过调用 setsockopt()来取消阻塞来自其他任务的 select()调用。该功能可用于一些联网应用，需要在 RTCS 中监视套接字的数据和应用事件, 该功能用于 Websocket 协议的配置。如果套接字位于 exceptfds 且 exceptfds 由 select()监视，它将通过 exceptfds 返回该套

接字以响应另一个对于 setsockopt() SO_EXCEPTION 的任务调用。应用应使用 getsockopt() SO_EXCEPTION 来读取套接字选项值并清零。套接字选项值可用于提示在任务之间的不同应用事件。

表 7-2. 正在连接/已连接数据流套接字

正在连接/已连接数据流套接字	select()
已接收第一个 FIN（被动连接关闭）。远程主机先向该连接发送 FIN，然后开始正常的连接关闭。	返回值 > 0。套接字位于被监视的 readfds 或 writefds 数组。进一步 send()可能。进一步 recv()返回-1。
已发送第一个 FIN（主动连接关闭）。RTCS 先向该连接发送 FIN，然后开始正常的连接关闭。	返回值 > 0。 套接字位于被监视的 readfds 或 writefds 数组。进一步 recv()可能。进一步 send()返回-1。
已接收 FIN（在发送后）。正常连接关闭完成。	返回值 > 0。套接字位于被监视的 readfds 或 writefds 数组。进一步 send()/recv()返回-1。
已发生 FIN（在接收后）。正常连接关闭完成。	返回值 > 0。 套接字位于被监视的 readfds 或 writefds 数组。进一步 send()或 recv()返回-1。
终止请求（远程主机向我们发送 RST 或不期望的 SYN）。	返回值 = -1。 RTCS_errno = RTCSERR_SOCK_ESHUTDOWN。 套接字位于被监视的 readfds 或 writefds 数组。 send()/recv()返回-1。 应用调用 closesocket()函数来释放由该套接字分配的资源。
终止请求（RTCS 发送 RST 以响应应用程序 closesocket()调用，从而导致连接终止）。	返回值 = -1。 RTCS_errno = RTCSERR_SOCK_CLOSED。 套接字句柄无效，应用程序不能使用该套接字。
数据段可供读取	返回值 > 0。 Readfds 具有可供读取数据的套接字句柄。
发送缓冲区空（远程主机应答了所有发送数据）	返回值 > 0。 Writefds 具有可以接受更多发送数据的套接字句柄。

表 7-3. 监听数据流套接字

监听数据流	select()
应用调用 shutdownsocket()	返回值 = -1。 RTCS_errno = RTCSERR_SOCK_ESHUTDOWN。 套接字位于被监视的 readfds 或 writefds 数组。
应用调用 closesocket()	返回值 = -1。 RTCS_errno = RTCSERR_SOCK_CLOSED。 套接字句柄无效，应用程序不能使用该套接字。
请求新连接	返回值 > 0。 Readfds 具有可以 accept()连接请求的套接字句柄。

表 7-4. 数据报套接字

数据报套接字	select()
应用程序调用 shutdownsocket()	不导致 select()返回。
应用程序调用 closesocket()	返回值 = -1。 RTCS_errno = RTCSERR_SOCK_CLOSED。 数据报套接字无效。
已接收数据报。	返回值 > 0。 Readfds 具有可供读取的已接收数据报的套接字句柄。

返回值

- select()函数返回包含在描述符集中的就绪套接字数量,或返回 RTCS_ERROR 如果发生错误。适当地设置 RTCS_errno。如果超出时限， select()返回 0。

另请参见

- RTCS_FD_SET
- RTCS_FD_ZERO
- RTCS_FD_CLR
- RTCS_FD_ISSET

示例

```
uint32_t socklist[3];
rtcs_fd_set rfds;
int32_t err;

socklist[0] = socket(AF_INET, SOCK_STREAM, 0);
socklist[1] = socket(AF_INET6, SOCK_STREAM, 0);
socklist[2] = socket(AF_INET, SOCK_DGRAM, 0);

…… /* call listen & bind as needed */

while(1)
{
  RTCS_FD_ZERO(&rfds);
  for(i=0; i<3; i++)
  {
     RTCS_FD_SET(socklist[i], &rfds);
  }

  err = select(3, &rfds, NULL, NULL, 0);
  if(RTCS_ERROR == err)
  {
     /* error occured */
  }
  else if(0 == err)
  {
    /* timeout */
  }
```

```
      else
      {
        if(FD_ISSET(socklist[0], &rfds))
        {
           …
        }
        if(FD_ISSET(socklist[1], &rfds))
        {
          …
        }
        if(FD_ISSET(socklist[2], &rfds))
        {
           …
      }
   }
}
```

7.158 RTCS_FD_SET

向 rtcs_fd_set 添加套接字。

概要

void RTCS_FD_SET(const uint32_t sock, rtcs_fd_set * const p_fd_set);

7.159 RTCS_FD_CLR

从 rtcs_fd_set 删除套接字。

概要

void RTCS_FD_CLR(const uint32_t sock, rtcs_fd_set * const p_fd_set);

7.160 RTCS_FD_ZERO

清除 rtcs_fd_set。

概要

void RTCS_FD_ZERO(rtcs_fd_set * const *p_fd_set*);

7.161 RTCS_FD_ISSET

检测套接字描述符是否位于 rtcs_fd_set 中。

概要

bool RTCS_FD_ISSET(const uint32_t *sock*, const rtcs_fd_set * const *p_fd_set*);

返回值

- 返回 TRUE，如果套接字位于**p_fd_set* 中，否则返回 FALSE。

编译时间选项

- RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT
  - 添加对于传统 RTCS_selectall()和 RTCS_selectset()函数的支持。
- RTCSCFG_SOCKET_OWNERSHIP
  - 除了 RTCS_selectall()/RTCS_selectset()，添加对于传统函数和结构的支持。

另请参见

- SOCK_Add_owner()
- SOCK_Remove_owner()
- SOCK_Is_owner()
- RTCS_attachsock()
- RTCS_detachsock()
- RTCS_transfersock()
- SOCK_OWNER_STRUCT

7.162 send()

向指定的远程终端发送数据流套接字或数据报套接字上的数据。

概要

```
int32_t     send(
            uint32_t     socket,
            char * buffer,
            uint32_t     buflen,
            uint32_t     flags)
```

参数

socket [in] — 用于要发送数据的套接字的句柄。

buffer [in] — 指向要发送数据的缓冲区的指针。

buflen [in] — 缓冲区中的字节数（无限制）。

flags [in] — 对于数据报套接字：从三个独立组中选择的基础协议的标志。只从后续 RTCS6_if_add_dns_addr () 中说明的一个或多个组中执行一个位宽或一个标志。对于流数据套接字，标志可以具有以下值：0、MSG_DONTWAIT 和 MSG_WAITACK。

说明

函数 send()向指定的远程终端发送数据流套接字或数据报套接字上的数据。

数据流套接字

RTCS 将数据（位于缓冲区）打包到 TCP 套接字并可靠、有序地将这些数据包发送到已连接的远程端点。

send()函数的返回时间取决于标志参数。

RTCS 向所有套接字附加一个推送标志用于发送缓冲区。所有数据立即发送，需要考虑远程端点缓冲区的容量。

数据报套接字

如果通过 connect()指定了远程端点，send()和 sendto()均使用指定的远程端点。如果未指定远程端点，send()返回 RTCS_ERROR。

标志参数覆盖是临时的，仅对当前调用的 send()有效。当向多个目的地广播或组播一个数据报时，将标志设定为 RTCS_MSG_NOLOOP 非常有用。当标志设定为 RTCS_MSG_NOLOOP 时，数据报不对本地主机接口复用。

数据报套接字的标志：

组 1:

- RTCS_MSG_BLOCK — 覆盖 OPT_SEND_NOWAIT 数据报套接字选项，并假设为 FALSE 进行操作。
- RTCS_MSG_NONBLOCK — 覆盖 OPT_SEND_NOWAIT 数据报套接字选项，并假设为 TRUE 进行操作。

组 2:

- RTCS_MSG_CHKSUM — 覆盖 OPT_CHECKSUM_BYPASS 校验和旁路选项，并假设为 FALSE 进行操作。
- RTCS_MSG_NOCHKSUM — 覆盖 OPT_CHECKSUM_BYPASS 校验和旁路选项，并假设为 TRUE 进行操作。

组 3:

- RTCS_MSG_NOLOOP — 不向环回接口发送数据报。
- 0 — 忽略。

数据流套接字的标志：

- 0 — 应用的套接字选项 SEND_NOWAIT。该选项的默认值为 FALSE。
- MSG_DONTWAIT — send()假设 SEND_NOWAIT 套接字选项为 TRUE 进行操作。
- MSG_WAITACK — send()直接使用应用程序数据缓冲区（不将数据复制到套接字的内部发送缓冲区中）并阻塞。

下表显示了对于数据流套接字，send()函数的返回时机：

标志参数	SEND_NOWAIT 套接字选项	send()的返回时机
零（0）	FALSE（默认）	阻塞。当所有数据传递到套接字的内部发送缓冲区时返回。
零（0）	TRUE	将所有数据（buflen 的最大值）复制到套接字的内部发送缓冲区并立即返回。
MSG_DONTWAIT	无关	将所有数据（buflen 的最大值）复制到套接字的内部发送缓冲区并立即返回。
MSG_WAITACK	无关	阻塞。当所有数据发送并由远程对等端应答后返回。

返回值

- 发送的字节数（成功）
- RTCS_ERROR（失败）

如果函数返回 RTCS_ERROR，应用可以调用 RTCS_geterror()来确定错误原因。

另请参见

- listen()

示例：数据流套接字

```
uint32_t    handle;
char        buffer[20000];
uint32_t    count;

...
count = send(handle, buffer, 20000, 0);
if (count == RTCS_ERROR)
 printf("\nError, send() failed with error code %lx",
        RTCS_geterror(handle));
```

7.163 sendto()

发送数据报套接字上的数据。

概要

```
int32_t   sendto(
          uint32_t              socket,
          char         *       buffer,
          uint32_t              buflen,
          uint16_t                flags,
          sockaddr     *       destaddr,
          uint16_t               addrlen)
```

参数

socket [in] — 用于要发送数据的套接字的句柄。

buffer [in] — 指向要发送数据的缓冲区的指针。

buflen [in] — 缓冲区中的字节数（无限制）。

flags [in] — 从三个独立组中选择的基础协议的标志。只从 RTCS6_if_add_dns_addr () 中描述的一个或多个组中执行一个按位或一个标志。

说明

函数向远程端点（位于 destaddr）发送作为 UDP 数据报的数据（位于缓冲区）。

还可以当通过 connect()预先指定远程端点时使用该函数。即使与预先指定的远程端点不同，还是会向 destaddr 发送数据报。

如果预先指定了套接字地址，您可以调用 sendto()，同时 destaddr 设为 NULL 和 addrlen 等于零：将该组合发送至预先指定的地址。调用 sendto()，同时 destaddr 设为 NULL 和 addrlen 等于零，但无预先指定目的地时将导致错误。

覆盖是临时的，仅对当前调用的 sendto()有效。当向多个目的地广播或组播一个数据报时，将标志设定为 RTCS_MSG_NOLOOP 非常有用。当标志设定为 RTCS_MSG_NOLOOP 时，数据报不对本地主机接口复用。

如果函数返回 RTCS_ERROR，应用可以调用 RTCS_geterror()来确定错误原因。

、该函数阻塞，但是会立即执行并回复命令。

返回值

- 发送的字节数（成功）
- RTCS_ERROR（失败）

示例

a) 向 IP 地址 192.203.0.54 发送 500 字节的数据，端口号 678。

```
        uint32_t       handle;
        sockaddr_in   remote_sin;
        uint32_t       count;
        char          my_buffer[500];
        ...
        for (i=0; i < 500; i++) my_buffer[i]= (i & 0xff);
        memset((char *) &remote_sin, 0, sizeof(sockaddr_in));

        remote_sin.sin_family = AF_INET;
        remote_sin.sin_port = 678;
        remote_sin.sin_addr.s_addr = 0xC0CB0036;

        count = sendto(handle, my_buffer, 500, 0, (struct sockaddr *)&remote_sin,
sizeof(sockaddr_in));
        if (count != 500)
            printf("\nsendto() failed with count %ld and error %lx",
                   count, RTCS_geterror(handle));
```

b) 向 FE80::2e0:4cFF:FE68:2343 发送"Hello, world!"，端口 7007，使用 IPv6 UDP 协议。

```
uint32_t socket_udp;
struct addrinfo    *foreign_addrv6_res /* pointer to PC IPv6 address */
struct addrinfo    *local_addrv6_res;  /* pointer to Board IPv6 address */
struct addrinfo    hints;              /* hints used for getaddrinfo() */
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_NUMERICHOST|AI_CANONNAME;
getaddrinfo ( "FE80::0200:5EFF:FEA8:0016%2", "7007", &hints, &local_addrv6_res);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_NUMERICHOST|AI_CANONNAME;
getaddrinfo ( "FE80::2e0:4cFF:FE68:2343","7007", &hints,&foreign_addrv6_res);
socket_udp = socket(AF_INET6, SOCK_DGRAM, 0);
error = bind(socket_udp, (sockaddr *)(local_addrv6_res->ai_addr), sizeof(struct
sockaddr_in6));
sendto(socket_udp,"Hello,world!", 13, 0, (sockaddr*)(foreign_addrv6_res->ai_addr),
sizeof(sockaddr_in6));
```

7.164 setsockopt()

设置套接字选项值。

概要

```
     uint32_t    setsockopt(
              uint32_t     socket,
              uint32_t     level,
              uint32_t     optname,
              void        *optval,
              uint32_t     optlen)
```

参数

socket [in] - 要改变选项的套接字的句柄。

level [in] — 选项所在的协议层:

SOL_IGMP

SOL_LINK

SOL_SOCKET

SOL_TCP

SOL_UDP

SOL_IP

SOL_IP6

optname [in] — 选项名称。

optval [in] — 指向选项值的指针。

optlen [in] — optval 指向的字节数。

返回值

- RTCS_OK（成功）
- 特定错误代码（失败）

另请参见

- ip_mreq

说明

您可以通过调用 setsockopt()设置大部分套接字选项。然而，不能用于设置以下选项。您只能通过 getsockopt()使用这些选项:

- IGMP 获取成员
- 可供读取的已接收字节数
- 接收以太网 802.1Q 优先级标签
- 接收以太网 802.3 帧
- 套接字错误
- 套接字类型

用户可变选项具有默认值。如果您想改变一些选项的值，您必须在绑定套接字之前完成。对于其他选项，您可以在套接字创建后的任意时刻进行修改。

该函数阻塞，但是会立即执行并回复命令。

注

一些选项可以临时针对套接字进行覆盖。欲了解更多信息，请参见 send()、sendto()、recv()和 recvfrom()。

选项

本节描述了套接字选项。

用于套接字的软件异常

选项名称	*SO_EXCEPTION*
协议层	*SOL_SOCKET*
值	*>= 0*
默认值	零
更改	任意时间
套接字类型	数据报或数据流
备注	RTCS 专用选项（不可移植）。setsockopt()用于设置选项值并通过第三个 rtcs_fd_set（exceptfds）参数使 select()返回，基于假设套接字设置在 exceptfds 中。getsockopt()用于从套接字读取选项值并将该选项值清零。

接收超时

选项名称	*SO_RCVTIMEO*
协议层	*SOL_SOCKET*
值	>= 零毫秒
默认值	0
更改	任意时间
套接字类型	数据报或数据流
备注	当超时过期时，无论是否接收了数据，recv()或 recvfrom()将立即返回。

校验和旁路

选项名称	*OPT_CHECKSUM_BYPASS*（可以覆盖）
协议层	*SOL_UDP*
值	• TRUE（RTCS 将发送数据报数据包的校验和字段设为零，并旁路校验和生成）。 • FALSE（RTCS 生成用于发送数据报数据包的校验和）。
默认值	FALSE
更改	在绑定之前
套接字类型	数据报
备注	—

连接超时

选项名称	*OPT_CONNECT_TIMEOUT*
协议层	*SOL_TCP*
值	≥ 0（RTCS 保持该毫秒数的连接）。
默认值	180,000（三分钟）。
更改	在绑定之前
套接字类型	数据流
备注	连接超时对应于 R2（在 RFC 793 中定义），有时称为硬超时。这指示了 RTCS 在放弃之前尝试建立连接所花的时间，对于已建立的连接，这指示了 RTCS 在确认连接中断前等待发送段码应答的时间。 如果远程端点在连接超时前不应答已发送的段码（例如，电缆断裂时会发生这种情况），RTCS 会关闭套接字连接和所有使用连接返回的函数调用。 RTCS 允许用户将该超时配置为任意值，从而应用可以快速检测到连接中断。

接收等待/无等待

选项名称	*OPT_RECEIVE_NOWAIT*
协议层	*SOL_UDP*
值	• TRUE（无论是否出现要接收的数据，recv()和 recvfrom()均立即返回）。 • FALSE（recv()和 recvfrom()等待直到出现要接收的数据）或发生接收超时过期。
默认值	FALSE
更改	任意时间
套接字类型	数据报
备注	—

IGMP 添加成员

选项名称	*RTCS_SO_IGMP_ADD_MEMBERSHIP*
协议层	*SOL_IGMP*
值	—
默认值	不在一个组内
更改	任意时间
套接字类型	数据报

备注	IGMP 必须位于 RTCS 协议表中。
	要加入一个组播组：
	`uint32_t                sock;`
	`struct  ip_mreq         group;`
	`group.imr_multiaddr.s_addr = multicast_ip_address;`
	`group.imr_interface.s_addr = local_ip_address;`
	`error = setsockopt(sock, SOL_IGMP,`
	`   RTCS_SO_IGMP_ADD_MEMBERSHIP, &group,`
	`   sizeof(group));`

IGMP 丢弃成员

选项名称	*RTCS_SO_IGMP_DROP_MEMBERSHIP*
协议层	*SOL_IGMP*
值	—
默认值	不在一个组内
更改	在套接字创建之后
套接字类型	数据报

备注	IGMP 必须位于 RTCS 协议表中。
	要脱离一个组播组:
	`uint32_t sock;`
	`struct ip_mreq group;`
	`group.imr_multiaddr.s_addr = multicast_ip_address;`
	`group.imr_interface.s_addr = local_ip_address;`
	`error = setsockopt(sock, SOL_IGMP,`
	`RTCS_SO_IGMP_DROP_MEMBERSHIP, &group,`
	`sizeof(group));`

IGMP 获取成员

选项名称	*RTCS_SO_IGMP_GET_MEMBERSHIP*
协议层	SOL_IGMP
值	—
默认值	不在一个组内
更改	—（仅用于 getsockopt()，返回值位于 optval 中）。
套接字类型	数据报
备注	—

初始重新传输超时

选项名称	*OPT_RETRANSMISSION_TIMEOUT*
协议层	*SOL_TCP*
值	>= *15 ms*（见备注）
默认值	3000（三秒）
更改	在绑定之前
套接字类型	数据流
备注	数值是预先估算、最佳的数据流套接字数据包所需的整个时间。如果 RTCS 在这段时间内未收到应答，它会尝试重新发送数据包。在连接建立后，RTCS 会从该初始值开始确定重新传输超时。如果初始重新传输超时不比套接字上预期的端到端响应时间长，则连接超时将提早过期。

保持活动超时

选项名称	*OPT_KEEPALIVE*

下一页继续介绍此表...

协议层	*SOL_TCP*
值	• 零（RTCS 不探测远程端点）。 • 非零（如果连接为空闲，RTCS 会周期性探测远程端点，这是用于检测远程端点是否仍然存在的操作）。
默认值	零分钟
更改	在绑定之前
套接字类型	数据流
备注	该选项不是 TCP/IP 规范的标准特性，会产生不必要的周期性网络流量。

最大重新传输超时

选项名称	*OPT_MAXRTO*
协议层	*SOL_TCP*
值	• 非零（用于重新传输定时器的指数补偿的最大值）。 • 零（RTCS 使用默认值，是段码最长寿命[MSL]的两倍）。由于 MSL 为 2 分钟，则 MTO 为 4 分钟）
默认值	零毫秒
更改	在绑定之前
套接字类型	数据流
备注	重新传输定时器用于一个段码的多次重新传输。

非 Nagle 算法

选项名称	*OPT_NO_NAGLE_ALGORITHM*
协议层	*SOL_TCP*
值	• TRUE（RTCS 不使用 Nagle 算法来合并短段码）。 • FALSE（用来减少网络拥塞，RTCS 使用 Nagle 算法[在 RFC 896 中定义]来合并短段码）。
默认值	FALSE
更改	在绑定之前
套接字类型	数据流
备注	如果应用程序尝试发送短段码，它将通过设置选项为 TRUE 来提高效率。

接收以太网 802.1Q 优先级标签

选项名称	*RTCS_SO_LINK_RX_8021Q_PRIO*
协议层	*SOL_LINK*
值	• -1（上一个接收帧不具有以太网 802.1Q 优先级标签）。 • 0..7（上一个接收帧具有以太网 802.1Q 优先级标签的指定优先级）。

下一页继续介绍此表...

默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）。
套接字类型	数据流（以太网）
备注	返回套接字接收的上一个帧的信息。

接收以太网 802.1Q VLAN 标识符标签

选项名称	*RTCS_SO_LINK_RX_8021Q_VID*
协议层	SOL_LINK
值	• -1（上一个接收帧不具有以太网 802.1Q VLAN 标识符标签）。 • 0…4094（上一个接收帧具有以太网 802.1Q 标签指定的 VLAN ID）。
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）
套接字类型	数据报或数据流（以太网）
备注	返回套接字接收的上一个帧的信息。

发送以太网 802.1Q VLAN 标识符标签

选项名称	*RTCS_SO_LINK_RX_8021Q_VID*
协议层	SOL_LINK
值	• -1（RTCS 不包括以太网 802.1Q 优先级标签）。 • 0…4094（RTCS 包括以太网 802.1Q 标签指定的 VLAN ID）。
默认值	-1
更改	任意时间
套接字类型	数据报或数据流（以太网）
备注	—

接收以太网 802.3 帧

选项名称	*RTCS_SO_LINK_RX_8023*
协议层	*SOL_LINK*
值	• TRUE（上一个接收帧为 802.3 帧）。 • FALSE（上一个接收帧为以太网 II 帧）。
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）
套接字类型	数据流（以太网）
备注	返回套接字接收的上一个帧的信息。

接收无等待

选项名称	*OPT_RECEIVE_NOWAIT*
协议层	*SOL_TCP*
值	• TRUE（无论是否有数据要接收，recv()总是立即返回）。 • FALSE（recv()等待直到接收到数据）或发生接收超时过期。
默认值	FALSE
更改	任意时间
套接字类型	数据流
备注	—

接收推送

选项名称	*OPT_RECEIVE_PUSH*
协议层	*SOL_TCP*
值	• TRUE（如果从远程端点接收到推送标志，则即使指定接收缓冲区未满，recv()仍然立即返回）。 • FALSE（recv()忽略推送标志，并且只有当缓冲区满或发生接收超时过期时才返回。
默认值	TRUE
更改	任意时间
套接字类型	数据流
备注	—

接收超时

选项名称	*OPT_RECEIVE_TIMEOUT*
协议层	*SOL_TCP*
值	• 零（在调用 recv()期间，RTCS 无期限等待输入数据）。 • 非零（在调用 recv()期间，对于输入数据，RTCS 等待指定毫秒数时间）。
默认值	零毫秒
更改	任意时间
套接字类型	数据流
备注	当超时过期时，无论是否接收了数据，recv()均将返回。该套接字选项在数据流套接字的 SOL_SOCKET 层上与 SO_RCVTIMEO 相同。

接收缓冲区大小 - 数据流套接字

选项名称	*OPT_RBSIZE*
协议层	*SOL_TCP*
值	建议为多个最大段码大小，最少为 3 个。
默认值	4380 字节
更改	在绑定之前

下一页继续介绍此表...

套接字类型	数据流
备注	当绑定套接字时，RTCS 为接收缓冲区分配指定的字节数，从而控制了 RTCS 可以为套接字缓冲的接收数据大小。

接收缓冲区大小 - 数据报套接字

选项名称	*OPT_RBSIZE*
协议层	*SOL_UDP*
值	一个 *UDP* 数据报中的最小数据大小。
默认值	4096 个字节
更改	任意时间
套接字类型	数据报
备注	UDP 套接字最多列队输入该字节数的数据报。

发送以太网 802.1Q 优先级标签

选项名称	*RTCS_SO_LINK_TX_8021Q_PRIO*
协议层	*SOL_LINK*
值	• -1（RTCS 不包括以太网 802.1Q 优先级标签） • 0..7（RTCS 包括以太网 802.1Q 优先级标签的指定优先级）
默认值	-1
更改	任意时间
套接字类型	数据流（以太网）
备注	—

发送以太网 802.3 帧

选项名称	*RTCS_SO_LINK_TX_8023*
协议层	*SOL_LINK*
值	• TRUE（RTCS 发送 802.3 帧）。 • FALSE（RTCS 发送以太网 II 帧）。
默认值	FALSE
更改	任意时间
套接字类型	数据流（以太网）
备注	返回套接字接收的上一个帧的信息。

发送无等待（数据报套接字）

选项名称	*OPT_SEND_NOWAIT*（可以覆盖）

下一页继续介绍此表...

协议层	*SOL_UDP*
值	• TRUE（RTCS 缓冲每个数据报且 send()或 sendto()立即返回）。 • FALSE（任务调用 send()或 sendto()阻塞，直到数据报发送完成；不复制数据报）。
默认值	FALSE
更改	任意时间
套接字类型	数据报
备注	—

发送无等待（数据流套接字）

选项名称	*OPT_SEND_NOWAIT*（可以覆盖）
协议层	*SOL_TCP*
值	• TRUE（如果数据正在等待发送，任务调用 send()不等待；RTCS 缓冲输出数据，并且 send()立即返回）。 • FALSE（如果数据正在等待发送，任务调用 send()等待）。
默认值	FALSE
更改	任意时间
套接字类型	数据流
备注	—

发送推送

选项名称	*OPT_SEND_PUSH*
协议层	*SOL_TCP*
值	• TRUE（如果可能，RTCS 向与 send()相关的段码中的最后一个数据包添加发送推送标志并立即发送数据。调用 send()可能阻塞直到另一个任务对该套接字调用 send()。 • FALSE（在发送数据包前，RTCS 等待直到它从主机接收了足够的数据以完全填充该数据包）。
默认值	TRUE
更改	任意时间
套接字类型	数据流
备注	—

发送超时

选项名称	*OPT_SEND_TIMEOUT*
协议层	*SOL_TCP*
值	• 零（在调用 send()期间，RTCS 无期限等待输出数据）。 • 非零（在调用 send()期间，对于输入数据，RTCS 等待指定毫秒数时间）。
默认值	4 分钟

下一页继续介绍此表...

更改	任意时间
套接字类型	数据流
备注	当超时过期时，send()返回

发送缓冲区大小

选项名称	*OPT_TBSIZE*
协议层	*SOL_TCP*
值	建议为多个最大段码大小，最少为 3 个。
默认值	4380 字节
更改	在绑定之前
套接字类型	数据流
备注	当绑定套接字时，RTCS 为发送缓冲区分配指定的字节数，从而控制了 RTCS 可以为套接字缓冲的发送数据大小。

套接字错误

选项名称	*OPT_SOCKET_ERROR*
协议层	SOL_SOCKET
值	—
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）
套接字类型	数据报或数据流
备注	返回套接字的上一个错误。

套接字类型

选项名称	*OPT_SOCKET_TYPE*
协议层	*SOL_SOCKET*
值	—
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）
套接字类型	数据报或数据流
备注	返回套接字类型（*SOCK_DGRAM* 或 *SOCK_STREAM*）

可供读取的字节:

选项名称	*SO_RCVNUM*
协议层	*SOL_SOCKET*
值	-

下一页继续介绍此表...

默认值	-
更改	-（仅用于 getsockopt()）
套接字类型	数据报或数据流
备注	返回可从较高层读取的套接字接收缓冲区中已接收数据的字节数。

时间等待超时

选项名称	*OPT_TIMEWAIT_TIMEOUT*
协议层	*SOL_TCP*
值	> 零毫秒
默认值	段码最长寿命的两倍（为常数）。
更改	在绑定之前
套接字类型	数据流
备注	返回套接字接收的上一个帧的信息。

接收目标地址

选项名称	RTCS_SO_IP_RX_DEST
协议层	*SOL_IP*
值	—
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）。
套接字类型	数据报或数据流
备注	返回套接字接收的上一个帧的目标地址。

有效时间 - 接收

选项名称	RTCS_SO_IP_RX_TTL
协议层	*SOL_IP*
值	—
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中）。
套接字类型	数据报或数据流
备注	获取输入数据包的 TTL（有效时间）字段。返回套接字接收的上一个帧的信息。

服务类型 - 接收

选项名称	RTCS_SO_IP_RX_TOS
协议层	*SOL_IP*

下一页继续介绍此表...

值	—
默认值	—
更改	— (仅用于 getsockopt()；返回值位于 *optval* 中)。
套接字类型	数据报或数据流
备注	返回输入数据包的 TOS（服务类型）字段。返回套接字接收的上一个帧的信息。

服务类型 - 发送

选项名称	RTCS_SO_IP_TX_TOS
协议层	*SOL_IP*
值	uchar
默认值	0
更改	任意时间
套接字类型	数据报或数据流
备注	设置或获取输出数据包的 IPv4 TOS（服务类型）字段。

有效时间 - 发送

选项名称	RTCS_SO_IP_TX_TTL
协议层	*SOL_IP*
值	输出数据报中 IP 头文件的 TTL 字段
默认值	64
更改	任意时间
套接字类型	数据报或数据流
备注	设置或获取输出数据包的 TTL（有效时间）字段。

本地地址

选项名称	RTCS_SO_IP_LOCAL_ADDR
协议层	*SOL_IP*
值	—
默认值	—
更改	—（仅用于 getsockopt()，返回值位于 optval 中)。
套接字类型	数据报或数据流
备注	返回本地 IP 地址。

输出单播数据包的 IPv6 跳码限制

选项名称	RTCS_SO_IP6_UNICAST_HOPS

下一页继续介绍此表...

协议层	SOL_IP6
值	0-255
默认值	0
更改	任意时间
套接字类型	数据报或数据流
备注	该选项定义了输出单播 IPv6 数据包的跳码限制。 选项值默认设为 0。这意味着由本地 IPv6 路由器设定跳码限制，否则跳码限制值为 64。

输出组播数据包的 IPv6 跳码限制

选项名称	RTCS_SO_IP6_MULTICAST_HOPS
协议层	SOL_IP6
值	0-255
默认值	1
更改	任意时间
套接字类型	数据报
备注	该选项定义了输出组播 IPv6 数据包的跳码限制。 如果设为 0，这意味着由本地 IPv6 路由器设定跳码限制，否则跳码限制值为 64。

IPv6 添加成员

选项名称	RTCS_SO_IP6_JOIN_GROUP
协议层	SOL_IP6
值	ipv6_mreq
默认值	—
更改	任意时间
套接字类型	数据报

备注	• 组播侦听者发现（MLDv1）协议可由 RTCSCFG_ENABLE_MLD 配置参数使能。这可使组播流量仅在一个本地网络中发生。
	• 可在一个套接字中同时存在的 IPv6 组播的最大成员数，这由 RTCSCFG_IP6_MULTICAST_SOCKET_MAX 配置参数定义。
	• 可在一个套接字中同时存在的唯一 IPv6 组播的最大成员数，这由 RTCSCF-RTCSCFG_IP6_MULTICAST_ MAX 配置参数定义。
	
	要加入一个 IPv6 组播组：
	
	`int sock;`
	`struct ipv6_mreq group;`
	`…`
	`IN6_ADDR_COPY(&<multicast_ip_address>, &`
	`group.ipv6imr_multiaddr);`
	`group.ipv6imr_interface = 0; /* Chosen by stack.*/`
	`<error> = setsockopt(sock, SOL_IP6,`
	`RTCS_SO_IP6_JOIN_GROUP, &group,`
	`sizeof(group));`

IPv6 丢弃成员

选项名称	RTCS_SO_IP6_LEAVE_GROUP
协议层	SOL_IP6
值	ipv6_mreq
默认值	—
更改	任意时间
套接字类型	数据报
备注	要脱离一个 IPv6 组播组: int sock; struct ipv6_mreq group; … IN6_ADDR_COPY(&<multicast_ip_address>, & group.ipv6imr_multiaddr); group.ipv6imr_interface = 0; /* Chosen by stack.*/ <error> = setsockopt(sock, SOL_IP6, RTCS_SO_IP6_LEAVE_GROUP, &group, sizeof(group));

套接字残留

选项名称	*SO_LINGER*
协议层	*SOL_SOCKET*
值	指针指向 struct linger
默认值	l_onoff = 0, l_linger_ms = 0
更改	任意时间
套接字类型	数据报或数据流

备注	示例
	
	struct linger so_linger = {0}; /* l_onoff = 0; l_linger_ms = 0; */
	int32_t status;
	
	so_linger.l_onoff = 1; /* l_onoff = 1; l_linger_ms = 0; */
	status = setsockopt(sock, SOL_SOCKET, SO_LINGER, (const
	void*)&so_linger, sizeof(so_linger));
	if(status == RTCS_OK)
	{
	    status = closesocket(sock);
	}

套接字保持有效

选项名称	*SO_KEEPALIVE*
协议层	*SOL_SOCKET*
值	*TRUE* 或 *FALSE*

下一页继续介绍此表...

默认值	FALSE
更改	在绑定之前
套接字类型	数据流
备注	该选项使能套接字连接的保持有效探测。这些探测用于维护 TCP 连接，并经常测试连接以确保仍然有效。仅对以连接为导向的协议（TCP）有效。默认值为 FALSE（禁止保持有效功能）。

保持有效计数

选项名称	*TCP_KEEPCNT*
协议层	*SOL_TCP*
值	*>=0*
默认值	8
更改	在绑定之前
套接字类型	数据流
备注	当 SO_KEEPALIVE 选项使能时，TCP 向已空闲一定时间的连接发送长度为零的数据包。如果远程主机不响应保持有效数据包，则在确认连接中断前，TCP 会重新发送一定次数的保持有效数据包。该保持有效数据包的默认重发次数限制为 8。对于给定套接字，TCP_KEEPCNT 选项可用于影响该数值，并指定要发送的保持有效探测的最大次数。

保持有效空闲时间

选项名称	*TCP_KEEPIDLE*
协议层	*SOL_TCP*
值	*>=0* 秒
默认值	7200（2 个小时）
更改	在绑定之前
套接字类型	数据流
备注	当 SO_KEEPALIVE 选项使能时，TCP 向已空闲一定时间的连接发送长度为零的数据包。对于给定套接字，TCP_KEEPIDLE 选项可用于影响该时间，并指定在保持有效探测之间空闲多少秒。空闲时间默认值为 7200 秒（2 个小时）。

保持有效间隔时间

选项名称	*TCP_KEEPINTVL*
协议层	*SOL_TCP*
值	*>=0* 秒
默认值	75 秒
更改	在绑定之前
套接字类型	数据流
备注	当 SO_KEEPALIVE 选项使能时，TCP 向已空闲一定时间的连接发送长度为零的数据包。如果远程主机不应答该保持有效数据包，则 TCP 在一段时间后会重发该数据包。该重发时间间隔的默认值为 75 秒。对于给定套接字，TCP_KEEPINTVL 选项可用于影响该数值，并指定在重发保持有效数据包前要等待的秒数。

示例

修改发送推送选项为 *FALSE*

```
uint32_t  handle;
uint32_t  opt_length = sizeof(uint32_t);
uint32_t  opt_value = FALSE;
uint32_t  status;
...
status =  setsockopt(handle, 0, OPT_SEND_PUSH,
                      &opt_value, opt_length);
if (status != RTCS_OK)
  printf("\nsetsockopt() failed with error %lx", status);

status = getsockopt(handle, 0, OPT_SEND_PUSH,
                    &opt_value, (uint32_t* *)&opt_length);
if (status != RTCS_OK)
    printf("\ngetsockopt() failed with error %lx", status);
```

修改接收-无等待选项为 *TRUE*

```
uint32_t  handle;
uint32_t  opt_length = sizeof(uint32_t);
uint32_t  opt_value = TRUE;
uint32_t  status;
...
status = setsockopt(handle, 0, OPT_RECEIVE_NOWAIT,
                    &opt_value, opt_length);
if (status != RTCS_OK)
  printf("\nError, setsockopt() failed with error %lx", status);
```

修改校验和旁路选项为 *TRUE*

```
uint32_t  handle;
uint32_t  opt_length = sizeof(uint32_t);
uint32_t  opt_value = TRUE;
uint32_t  status;
...
status = setsockopt(handle, SOL_UDP, OPT_CHECKSUM_BYPASS,
                    &opt_value, opt_length);
if (status != RTCS_OK)
  printf("\nError, setsockopt() failed with error %lx", status);
```

修改发送 TTL

```
uint32_t  handle;
uint32_t  status;
uint8_t   opt_value = 64;
...
status = setsockopt(handle, SOL_IP, RTCS_SO_IP_TX_TTL,
                    (void *)&opt_value, sizeof(opt_value));
if (status != RTCS_OK)
   printf("\nError, setsockopt() failed with error %lx", status);
```

7.165 SOL_NAT_getsockopt

读取 NAT 配置。

概要

```
uint32_t SOL_NAT_getsockopt(
        uint32_t optname,
        void *optval,
        uint32_t *optlen)
```

参数

- optname [in] 选项名称, RTCS_SO_NAT_PORTS 或 RTCS_SO_NAT_TIMEOUTS 之一。
- optval [in/out] 指向要写入配置位置的指针。
- optlen [in/out] 指向字节数的指针。在函数输入时，该数值用于确定 optval 参数指向的数据结构的大小。在函数返回时，为实际的写入字节数。

返回值

- 零 (成功 - RTCS_OK)
- 非零 (失败 - 特定错误代码)

示例

```
#include <nat.h>

nat_ports ports;
nat_timeouts nat_touts;
uint32_t error;
uint32_t optlen;

optlen = sizeof(ports);
error = SOL_NAT_getsockopt(RTCS_SO_NAT_PORTS, &ports, &optlen);

optlen = sizeof(nat_touts);
error = SOL_NAT_getsockopt(RTCS_SO_NAT_TIMEOUTS, &nat_touts, &optlen);
```

7.166 SOL_NAT_setsockopt

配置 NAT。

概要

```
uint32_t SOL_NAT_setsockopt(
           uint32_t optname,
```

```
          void *optval,
          uint32_t optlen)
```

参数

- optname [in] 选项名称，RTCS_SO_NAT_PORTS 或 RTCS_SO_NAT_TIMEOUTS 之一。
- optval [in] 指向选项值的指针。
- optlen [in] optval 指向的字节数。

返回值

- 零（成功 - RTCS_OK）
- 非零值（失败 - 特定错误代码）

示例：通过 RTCS NAT 将最大端口数改为 30000，不改变最小端口数：

```
#include <nat.h>

nat_ports ports;
uint32_t error;

ports.port_min = 0; /* No modification */
ports.port_max = 30000;

error = SOL_NAT_setsockopt(RTCS_SO_NAT_PORTS, &ports, sizeof(ports));
```

示例：修改 TCP 和 UDP 非活动超时值，不改变 FIN 超时值：

```
#include <nat.h>

nat_timeouts nat_touts;
uint32_t error;

nat_touts.timeout_tcp = 700000; /* Time in milliseconds */
nat_touts.timeout_udp = 500000; /* Time in milliseconds */
nat_touts.timeout_fin = 0;      /* No modification */

error = SOL_NAT_setsockopt(RTCS_SO_NAT_TIMEOUTS, &nat_touts, sizeof(nat_touts));
```

7.167 shutdownsocket()

禁止在套接字上发送和/或接收。

概要

```
int32_t shutdownsocket(uint32_t sock, int32_t how);
```

参数

- sock [in] – 套接字句柄
- how [in] – 套接字关闭类型

说明

禁止在数据报或数据流套接字上发送和/或接收。对于数据流套接字，关闭所有或部分的全双工连接。how 参数指定了关闭类型。可能的值如下:

- SHUT_RD 将不允许后续接收。
- SHUT_WR 将不允许后续发送。这可能导致发生特定于套接字协议系列的操作。
- SHUT_RDWR 将不允许后续发送和接收。包含 SHUT_WR 协议的特定操作。

根据套接字属性，以下协议特定操作将应用到 SHUT_WR （也可能是 SHUT_RDWR）的使用。

域	类型	协议	返回值和操作
AF_INET	SOCK_DGRAM	UDP	返回 0
AF_INET	SOCK_STREAM	TCP	返回 0。发送队列数据、等待 ACK，然后发送 FIN。
AF_INET6	SOCK_DGRAM	UDP	返回 0
AF_INET6	SOCK_STREAM	TCP	返回 0。发送队列数据、等待 ACK，然后发送 FIN。

当不允许后续发送且应用尝试发送 send()/sendto()时, send()/sendto()返回-1 并且套接字中的错误代码将设为 RTCSERR_SOCK_ESHUTDOWN。当不允许后续接收且应用尝试发送 recv()/recvfrom()时, recv()/recvfrom()返回-1 并且套接字中的错误代码将设为 RTCSERR_SOCK_ESHUTDOWN。当在正在监听的 TCP 套接字上调用 shutdownsocket()，select()和 accept()函数将返回-1 并且 RTCS_errno 将设为 RTCSERR_SOCK_ESHUTDOWN (假设 select()/accept()监视要关闭的套接字句柄)。可以在套接字上重复调用 shutdownsocket()。以下状态机图表显示了在同一个套接字上重复调用 shutdownsocket()的所支持的“how”参数顺序:

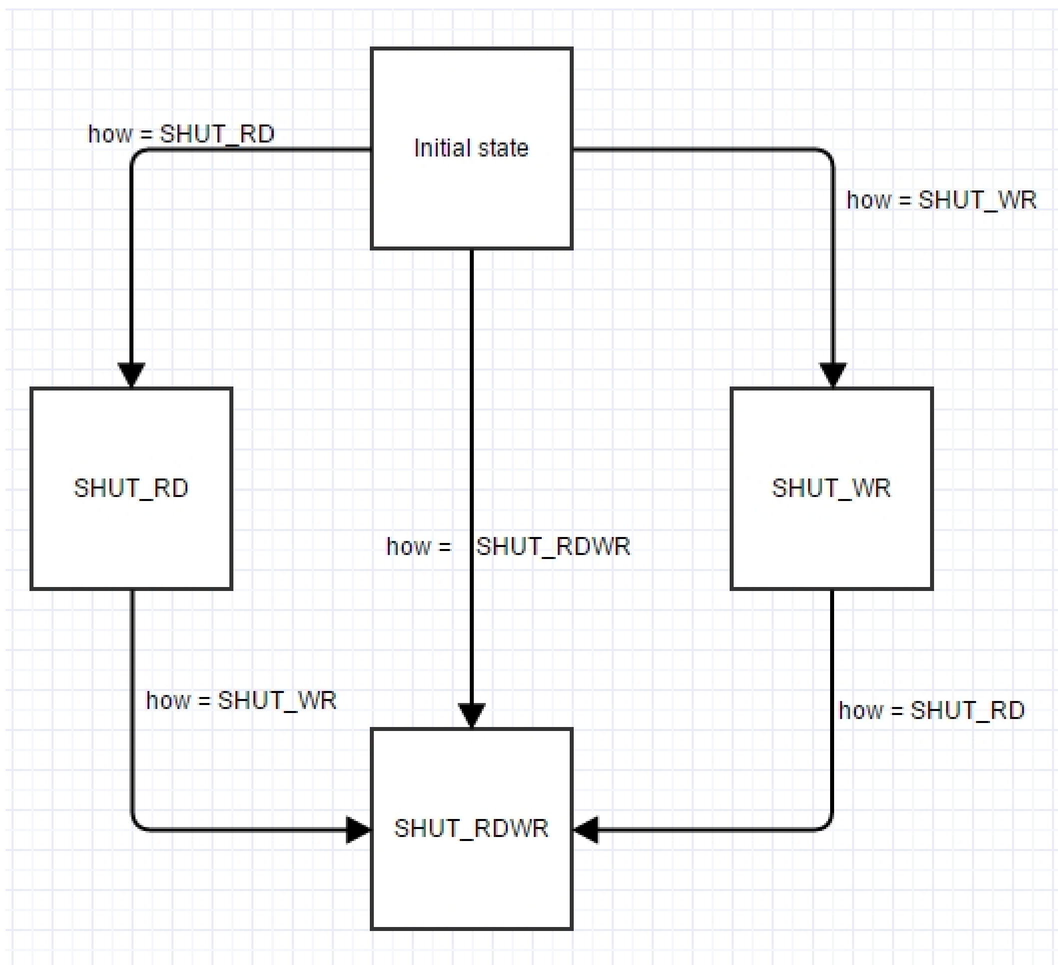

图 7-1. 状态机图表

当套接字的关闭状态为 SHUT_RDWR, 尝试 shutdownsocket()将只会返回 RTCS_OK 且不会改变内部套接字状态。

7.168 shutdown()

关闭套接字。该函数支持向下兼容性。在新工程中，建议使用 closesocket()和可选的 SO_LINGER 套接字选项。

概要

```
uint32_t  shutdown(
          uint32_t  socket,
          uint16_t  how)
```

参数

socket [in] — 要关闭的套接字句柄。

how [in] — 以下之一（见说明):

FLAG_CLOSE_TX

FLAG_ABORT_CONNECTION

说明

请注意，在调用 shutdown()后，应用将不再使用套接字。

shutdown()阻塞，但是命令继续处理且立即返回。

套接字类型	***how*** 的值	操作
数据报	忽略	• 立即关闭套接字。 • 调用 recvfrom()立即返回。 • 丢弃队列中的输入数据包。
取消连接数据流	忽略	立即关闭套接字。
已连接数据流	FLAG_CLOSE_TX	• 关闭套接字，确保所有发送数据均被应答。 • 调用 send()和 recv()立即返回。 • 如果 RTCS 引起断开连接，在远程端点关闭连接后，RTCS 会保持 4 分钟的内部套接字上下文（是最大 TCP 断码寿命的两倍）。
	FLAG_ABORT_CONNECTION	• 立即丢弃内部套接字上下文。 • 向远程端点发送 TCP 复位数据包。 • 调用 send()和 recv()立即返回。

返回值

- RTCS_OK
- 特定错误代码

示例

```
uint32_t handle;
uint32_t status;
...
status = shutdown(handle, 0);
if (status != RTCS_OK)
  printf("\nError, shutdown() failed with error code %lx",
        status);
```

7.169 SMTP_send_email

用于发送电子邮件的函数。

概要

```
_mqx_int SMTP_send_email(
SMTP_PARAM_STRUCT_PTR     param,
char                    *err_string,
uint32_t                  buffer_size)
```

参数

param [IN] — 指向包含所有所需参数的结构的指针。

err_string[OUT] — 指向用于发送消息/错误消息的用户缓冲区的指针。该参数可以为 *NULL* - 无消息，然后返回。

buffer_size[IN] — 参数 err_string 的字节大小。

说明

params 结构包含 SMTP 客户端所需的所有信息。这包括 SMTP 地址、电子邮件文本、用于发送电子邮件的服务器、登录账号和密码（只在需要验证时使用）。

返回值

- *SMTP_OK* — 电子邮件发送成功。
- *SMTP_ERR_BAD_PARAM* — 在 param 结构中设置的无效值。
- *SMTP_ERR_CONN_FAILED* — 连接服务器失败。
- *SMTP_WRONG_RESPONSE* — 服务器向 SMTP 命令返回错误的响应。
- *MQX_OUT_OF_MEMORY* — 用于 SMTP 客户端关键组件的存储器分配失败。

示例

关于用于演示函数 SMTP_send_email 用法的源程序，请参见文件\shell\source\rtcs\sh_smtp.c。

7.170 SNMP_init()

启动 SNMP 代理。

概要

```
uint32_t  SNMP_init(
          char *name,
          uint32_t   priority
          uint32_t   stacksize)
```

参数

name [in] — SNMP 代理任务的名称。

priority [in] — SNMP 代理任务的优先级 (我们建议您设置一个较高的数字，使其优先级低于 RTCS 任务的优先级)。

stacksize [in] — SNMP 代理任务的协议栈大小。

说明

该函数启动 SNMP 代理并创建 SNMP 任务。

返回值

- RTCS_OK （成功）
- 错误代码 （失败）

另请参见

- MIB1213_init()

示例

```
uint32_t error;
/* register the RFC1213 MIB */
MIB1213_init();
/* Start SNMP Agent: */
error = SNMP_init("SNMP agent", 7, 1000);
if (error)
         return error;
printf("\nSNMP Agent is running");
```

7.171 SNMP_trap_warmStart()

概要

```
void SNMP_trap_warmStart(void)
```

说明

该函数发送一个热启动陷阱类型 1/0。SNMP 陷阱版本为 1。

返回值

7.172 SNMP_trap_coldStart()

概要

```
void SNMP_trap_coldStart(void)
```

说明

该函数发送一个冷启动陷阱类型 0/0。SNMP 陷阱版本为 1。

返回值

7.173 SNMP_trap_authenticationFailure()

概要

```
void SNMP_trap_authenticationFailure(void)
```

说明

该函数发送一个验证失败陷阱类型 4/0。SNMP 陷阱版本为 1。

返回值

7.174 SNMP_trap_linkDown()

概要

```
void SNMP_trap_linkDown(void *ihandle)
```

参数

ihandle [in] — 接口索引

说明

该函数发送一个链路中断陷阱类型 2/0。SNMP 陷阱版本为 1。

返回值

7.175 SNMP_trap_myLinkDown()

概要

```
void SNMP_trap_myLinkDown(void *ihandle)
```

参数

ihandle [in] — 企业指定接口索引

说明

该函数发送一个用于企业指定设备的链路中断陷阱类型 2/0。SNMP 陷阱版本为 1。

返回值

7.176 SNMP_trap_linkUp()

概要

```
void SNMP_trap_linkUp(void *ihandle)
```

参数

ihandle [in] — 接口索引

说明

该函数发送一个链路连接陷阱类型 3/0。SNMP 陷阱版本为 1。

返回值

7.177 SNMP_trap_userSpec()

概要

```
    void SNMP_trap_userSpec(
            RTCSMIB_NODE_PTR trap_node,
            uint32_t spec_trap,
            RTCSMIB_NODE_PTR enterprises)
```

参数

trap_node [in] — 用户指定陷阱节点

spec_trap [in] — 用户指定陷阱类型

enterprises [in] — 企业节点

说明

该函数发送用户指定陷阱 6/spec_trap 类型 1 消息。

返回值

7.178 SNMPv2_trap_warmStart()

概要

```
void SNMPv2_trap_warmStart(void)
```

说明

该函数发送热启动陷阱类型 2 消息。

返回值

7.179 SNMPv2_trap_coldStart()

概要

```
void SNMPv2_trap_coldStart(void)
```

说明

该函数发送冷启动陷阱类型 2 消息。

返回值

另请参见

- SNMP_trap_coldStart()

7.180 SNMPv2_trap_authenticationFailure()

概要

```
void SNMPv2_trap_authenticationFailure(void)
```

说明

该函数发送验证失败陷阱类型 2 消息。

返回值

7.181 SNMPv2_trap_linkDown()

概要

```
void SNMPv2_trap_linkDown(void *ihandle)
```

参数

ihandle [in] — 接口索引

说明

该函数发送链路中断陷阱类型 2 消息。

返回值

7.182 SNMPv2_trap_linkUp()

概要

```
void SNMPv2_trap_linkUp(void *ihandle)
```

参数

ihandle [in] — 接口索引

说明

该函数发送链路连接陷阱类型 2 消息。

返回值

7.183 SNMPv2_trap_userSpec()

概要

```
void SNMPv2_trap_userSpec(
        RTCSMIB_NODE_PTR trap_node)
```

参数

trap_node [in] — 用户指定陷阱节点

说明

该函数发送用户指定陷阱类型 2 消息。

返回值

7.184 SNTP_init()

启动 SNTP 客户端任务。

概要

```
uint32_t SNTP_init(
char              *name,
uint32_t            priority,
uint32_t            stacksize,
_ip_address        destination,
uint32_t             poll)
```

参数

name [in] — SNTP 客户端任务的名称。

priority [in] — SNTP 客户端任务的优先级（我们建议您设置一个较高的数字，使其优先级低于 RTCS 任务的优先级）。

stacksize [in] — SNTP 客户端任务的协议栈大小。

destination [in] — 发送 SNTP 时间请求的目的地。以下之一：

- 时间服务器的 IP 地址（单播模式）。
- 本地组播地址或组播组（任播模式）。

poll [in] — 在时间更新期间的等待时间（必须位于 1 至 4294967 秒之间）。

说明

当函数启动 SNTP 客户端任务时，将先更新本地时间，然后等待由轮询指定的秒数。时间超时后，SNTP 客户端会重复相同的循环。在 UTC（协调世界时间）中设置本地时间。

SNTP 客户端任务工作在单播或任播模式下。

返回值

- RTCS_OK（成功）
- RTCSERR_INVALID_PARAMETER（失败）由未指定目的地或超出查询范围导致。
- 特定错误代码（失败），由调用 socket()和 bind()导致。

示例

```
uint32_t error;

/*
** Start the SNTP Client task with the following settings:
** Task Name: SNTP Client
** Priority: 7
** Stacksize: 1000
** Server address: 142.123.203.66 = 0x8E7BCB42
** Poll interval: every 100 seconds
*/

error = SNTP_init("SNTP client", 7, 1000, 0x8E7BCB42, 100);
if (error) return error;
printf("The SNTP client task is running");
return 0;
```

7.185 SNTP_oneshot()

通过 SNTP 协议，在 UTC 时间中设置该时间。

概要

```
uint32_t SNTP_oneshot(
         _ip_address  destination,
         uint32_t       timeout)
```

参数

destination [in] — 发送 SNTP 时间请求的目的地。以下之一:

- 时间服务器的 IP 地址（单播模式)。
- 本地组播地址或组播组（任播模式)。

timeout [in] — 通过 SNTP 持续尝试获取该时间的时间（以毫秒为单位)。

说明

该函数发送一个 SNTP 数据包并等待回答。如果在超时前收到回答，则设置时间。如果在指定时间内未收到回答，则返回 RTCSERR_TIMEOUT。在 UTC（协调世界时间）中设置本地时间。

SNTP 客户端任务工作在单播或任播模式下。

返回值

- RTCS_OK（成功)
- RTCSERR_INVALID_PARAMETER（失败）由未指定目的地导致。
- RTCSERR_TIMEOUT（失败）由于未在超时前通过 SNTP 成功接收时间导致。
- 错误代码（失败)。

7.186 socket()

创建套接字。

概要

```
uint32_t socket(
       uint16_t   protocol_family,
       uint16_t   type,
       uint16_t   protocol)
```

参数

protocol_family [in] — 协议系列。必须为 *PF_INET*（协议系列，IP 寻址）。

type [in] — 套接字类型。以下之一：

- SOCK_STREAM
- SOCK_DGRAM

protocol [in] — 未使用

说明

该应用使用套接字句柄来连续使用套接字。该函数阻塞，但是会立即执行并响应命令。

返回值

- 套接字句柄（成功）
- RTCS_SOCKET_ERROR（失败）

示例

请参见 bind()。

7.187 TCP_stats()

获取指向 TCP 统计数据的指针。

概要

```
TCP_STATS_PTR  TCP_stats(void)
```

说明

函数 TCP_stats()没有参数。它返回 RTCS 收集的 TCP 统计数据。

返回值

指向 TCP_STATS 结构的指针。

另请参见

- TCP_STATS

7.188 TFTPCLN_connect()

连接到 TFTP 服务器。

概要

```
uint32_t TFTPCLN_connect(
    TFTPCLN_PARAM_STRUCT *params)
```

参数:

- params[in] - TFTP 服务器的参数。

说明:

函数 TFTPCLN_client()根据_params_结构中的参数启动 TFTP 客户端。在成功调用该函数后，TFTP 客户端就可以向/从服务器发送/接收文件了。在该结构中，至少必须设置一个远程主机信息。欲了解每个服务器参数的更多说明，请参见第 X.Y 章 (链接到描述 TFTPCLN_PARAM_STRUCT 的章节)。

返回值:

- 非零值 (成功)
- 零 (失败)

示例:

```
#include "tftpcln.h"

struct addrinfo         hints = {0};
struct addrinfo         *getadd_result;
int                     error;
uint32_t                handle;
int32_t                 result;
TFTPCLN_PARAM_STRUCT    params = {0};

hints.ai_family = AF_UNSPEC;
error = getaddrinfo("192.168.1.1", "69", &hints, &getadd_result);
if (error == 0)
{
    params.sa_remote_host = *getadd_result->ai_addr;

    freeaddrinfo(getadd_result);

    handle = TFTPCLN_connect(&params);
    if (handle == 0)
    {
        printf("Error");
        _task_block();
    }
    else
    {
        printf("Downloading firmware from server...");
        result = TFTPCLN_get(handle, "a:\\firmware.bin", "firmware.bin");
        if (result == RTCS_OK)
```

```
            {
                printf("successful");
            }
            else
            {
                printf("failed");
            }
            TFTPCLN_disconnect(handle);
        }
    }
    else
    {
        printf("Failed to resolve remote host address");
    }
```

另请参见:

- TFTPCLN_disconnet()
- TFTPCLN_PARAM_STRUCT

7.189 TFTPCLN_get

从 TFTP 服务器下载文件。

概要

```
    int32_t TFTPCLN_get(
        uint32_t handle,
        char *local_file,
        char *remote_file)
```

参数:

- handle[in] - 由函数 TFTPCLN_connect()创建的 TFTP 客户端的句柄。该参数为必填项。
- local_file[in] - 将存放于本地的文件的文件名。该参数可为 NULL，则本地文件名与远程文件名一致。
- remote_file[in] - 要从服务器下载的文件的文件名。该参数为必填项（不能为 NULL）。

说明:

该函数用于通过 TFTP 协议将远程文件下载到本地系统。调用该函数阻塞，直到下载完成或失败。

返回值:

- RTCS_OK（成功）
- RTCS_ERROR（失败）

另请参见:

- TFTPCLN_put
- TFTPCLN_connect
- TFTPCLN_disconnect

7.190 TFTPCLN_put

上传文件到 TFTP 服务器。

概要

```
int32_t TFTPCLN_put(
    uint32_t handle,
    char *local_file,
    char *remote_file)
```

参数:

- handle[in] - 由函数 TFTPCLN_connect()创建的 TFTP 客户端的句柄。该参数为必填项。
- local_file[in] - 要从本地系统发送到服务器的文件的文件名。该参数为必填项(不能为 NULL)。
- remote_file[in] - 在服务器上创建的文件的文件名。该参数可为 NULL，则远程文件名与本地文件名一致。

说明:

该函数用于通过 TFTP 协议将本地文件上传到远程系统。调用该函数阻塞，直到下载完成或失败。

返回值:

- RTCS_OK （成功)
- RTCS_ERROR （失败)

另请参见:

- TFTPCLN_put
- TFTPCLN_connect
- TFTPCLN_disconnect

7.191 TELNETSRV_init

启动 Telnet 服务器。

概要

```
uint32_t TELNETSRV_init(
TELNETSRV_PARAM_STRUCT *params)
```

参数

params[in] - Telnet 服务器的参数。

说明:

函数 TELNETSRV_init()根据_params_结构中的参数启动 Telnet 服务器。

shell 函数和 shell 命令参数为必填项。如果未提供，任何已连接的客户端将立即断开连接。关于每个服务器参数的说明，请参见 TELNETSRV_PARAM_STRUCT 章节。

返回值

- 非零值（成功）
- 零（失败）

示例

```
#include "shell.h"
#include "telnetsrv.h"
extern const SHELL_COMMAND_STRUCT Telnetsrv_shell_commands[];
     TELNETSRV_PARAM_STRUCT params = {0};
     uint32_t handle;

     params.shell_commands = (void *) Telnetsrv_shell_commands;
     params.shell = (TELNET_SHELL_FUNCTION) Shell;
     handle = TELNETSRV_init(params);
```

另请参见

- TELNETSRV_release
- TELNETSRV_PARAM_STRUCT

7.192 TELNETSRV_release

停止 Telnet 服务器并释放其所有资源。

概要

```
uint32_t TELNETSRV_release(
uint3_t server_h)
```

参数

server_h[in] - 服务器句柄（来自函数 TELNETSRV_init）。

说明

该函数的操作与 TELNETSRV_init()相反。它关闭所有正在监听的套接字, 停止所有服务器任务并释放服务器使用的所有内存。将阻塞调用任务，直到服务器停止且资源释放为止。

返回值

- RTCS_OK - 成功关闭。
- RTCS_ERR - 关闭失败。

另请参见

- TELNETSRV_init

7.193 TFTPSRV_init

启动 TFTP 服务器。

概要

```
uint32_t TFTPSRV_init(
TFTPSRV_PARAM_STRUCT *params)
```

参数

params[in] - TFTP 服务器的参数。

说明

函数 TFTPSRV_init()根据_params_结构中的参数启动 TFTP 服务器。至少要在该结构中设置一个根目录。关于每个服务器参数的说明，请参见 TFTPSRV_PARAM_STRUCT。

返回值

- 非零值（成功）
- 零（失败）

示例

```
#include "tftpsrv.h"
    TFTPSRV_PARAM_STRUCT params = {0};
    uint32_t handle;

    params.root_dir = "a:";
    handle = FTPSRV_init(params);
```

另请参见

- TFTPSRV_release
- TFTPSRV_PARAM_STRUCT

7.194 TFTPSRV_release

停止 TFTP 服务器并释放其所有资源。

概要

```
uint32_t TFTPSRV_release(
uint32_t server_h)
```

参数

server_h[in] - 服务器句柄（来自函数 TFTPSRV_init)。

说明

该函数的操作与 TFTPSRV_init()相反。它关闭所有正在监听的套接字，停止所有服务器任务并释放服务器使用的所有内存。将阻塞调用任务，直到服务器停止且资源释放为止。

返回值

- RTCS_OK - 成功关闭。
- RTCS_ERR - 关闭失败。

另请参见

- TFTPSRV_init

7.195 UDP_stats()

获取指向 UDP 统计数据的指针。

概要

```
UDP_STATS_PTR  UDP_stats(void)
```

说明

- DHCPSRV_DATA_STRUCT

函数 UDP_stats()获取指向 UDP 统计数据的指针，该统计数据由 RTCS 收集。

返回值

指向 *UDP_STATS* 结构的指针。

另请参见

- ICMP_STATS
- IGMP_STATS
- TCP_STATS
- ARP_STATS

7.196 按服务列出函数

服务	函数
DHCP 客户端	RTCS_if_bind_DHCP()
DHCP 服务器	DHCP* DHCPSRV*
DNS 解析程序	getaddrinfo()
Echo 服务器	ECHOSRV_release()
以太网驱动程序	ENET_get_stats()（MQX RTOS 的组成部分） ENET_initialize() （MQX RTOS 的组成部分）
FTP 客户端	
FTP 服务器	FTPSRV_release()
HTTP 服务器	HTTPSRV_init() HTTPSRV_release() HTTPSRV_cgi_read() HTTPSRV_cgi_write() HTTPSRV_ssi_write()
IPCFG	ipcfg_bind_boot() ipcfg_bind_dhcp() ipcfg_add_interface() ipcfg_get_ihandle() ipcfg_get_mac() ipcfg_get_state()

下一页继续介绍此表...

服务	函数
	ipcfg_get_state_string() ipcfg_get_desired_state() ipcfg_get_link_active() ipcfg_add_dns_ip() ipcfg_del_dns_ip() ipcfg_get_ip() ipcfg_get_tftp_serveraddress() ipcfg_get_tftp_servername() ipcfg_get_boot_filename() ipcfg_poll_dhcp() ipcfg_task_create() ipcfg_task_destroy() ipcfg_task_status() ipcfg_task_poll()
IWCFG	iwcfg_set_essid() iwcfg_get_essid() iwcfg_commit() iwcfg_set_mode() iwcfg_get_mode() iwcfg_set_wep_key() iwcfg_get_wep_key() iwcfg_set_passphrase() iwcfg_get_passphrase() iwcfg_set_sec_type() iwcfg_get_sectype() iwcfg_set_power() iwcfg_set_scan()
MIB	MIB1213_init()
NAT	NAT_init() NAT_close() NAT_stats()
PPP 驱动程序	PPP_init() PPP_release() PPP_pause() PPP_resume()
RTCS	RTCS_if_add() RTCS_if_bind() RTCS_if_bind_BOOTP() RTCS_if_bind_DHCP()

下一页继续介绍此表...

按服务列出函数

服务	函数
	RTCS_if_bind_IPCP() RTCS_if_remove() RTCS_if_unbind() RTCS_ping() RTCSLOG_disable() RTCSLOG_enable()
SNMP 代理	SNMP_init() SNMP_trap_coldStart() MIB1213_init() MIB_find_objectname() MIB_set_objectname()
SNTP 客户端	
套接字	listen() RTCS_selectall() RTCS_selectset() select()
统计数据	IGMP_stats() NAT_stats() TCP_stats()
Telnet 客户端	TFTPCLN_connect()
Telnet 服务器	
TFTP 服务器	TFTPSRV_init

第 8 章
编译时选项

8.1 编译时选项

您可以通过更改编译时配置选项的值包含或去除 RTOS 附带的某些功能。如果您改变了一个值，您必须重新编译 RTCS。关于重新编译 RTCS 的更多信息，请参见第 6 章，“重新编译”。

与 PSP、BSP 或其他包含在 Freescale MQX RTOS 中的系统库类似，RTCS 编译工程从中央用户配置文件 user_config.h 中获取编译时配置选项。该文件位于顶层配置文件夹的开发板特定子目录中。

列出的所有配置宏及其默认值由 source\include\rtcscfg.h 文件定义。用户不应修改该文件。在 RTCS 编译工程中设置了合适的包含搜索路径，rtcscfg.h 文件包含来自开发板特定配置目录的 user_config.h 文件并针对给定开发板使用合适的配置选项。

若要:	设置选项值到:
包含该选项。	*1*
排除该选项。	*0*

8.2 推荐设置

选择用于编译时配置选项的设置应由应用的需求决定。表 8-1 向您展示了可用于开发应用的一些常用设置。

表 8-1. 推荐的编译时设置

选项	默认值	调试	速度	大小
RTCSCFG_IP_DISABLE_ **DIRECTED_BROADCAST**	0	0	0	0
RTCSCFG_ENABLE_8021Q	0	0，1	0，1	0，1

下一页继续介绍此表...

表 8-1. 推荐的编译时设置 (继续)

选项	默认值	调试	速度	大小
RTCSCFG_LINKOPT_8023	0	0，1	0，1	0，1
RTCSCFG_LOG_PCB	0	1	0	0
RTCSCFG_LOG_SOCKET_API	0	1	0	0

8.3 配置选项和默认设置

默认值在 rtcs/include/rtcscfg.h 中定义。您可以改写 user_config.h 用户配置文件中的设置。

8.3.1 RTCSCFG_FD_SETSIZE

每个 rtcs_fd_set 的套接字数量。默认为 8。

8.3.2 RTCSCFG_SOMAXCONN

用于监听 backlog 队列长度的系统最大宽度。默认为 5。

8.3.3 RTCSCFG_ARP_CACHE_SIZE

每个网络接口的 ARP 缓存表大小。

8.3.4 RTCSCFG_IP_DISABLE_DIRECTED_BROADCAST

默认情况下，RTCS 接收并直接传递广播数据报。将该值设为 1 以降低 Smurf ICMP 回应请求 DoS 攻击的风险。

8.3.5 RTCSCFG_BACKWARD_COMPATIBILITY_RTCSSELECT

添加对于传统 RTCS_selectall()和 RTCS_selectset()函数的支持。

8.3.6 RTCSCFG_BOOTP_RETURN_YIADDR

当 RTCSCFG_BOOTP_RETURN_YIADDR 为 1 时，BOOTP_DATA_STRUCT 的另一字段将填充 BOOTREPLY 的 YIADDR 字段。

8.3.7 RTCSCFG_DISCARD_SELF_BCASTS

默认情况下，控制是否丢弃我们发送的所有广播数据包，因为它们很可能是之前集线器的回应。

8.3.8 RTCSCFG_UDP_ENABLE_LBOUND_MULTICAST

当 RTCSCFG_UDP_ENABLE_LBOUND_MULTICAST 为 1 时，作为组播组成员的本地绑定套接字将可以接收发送到其单播和组播地址的消息。

8.3.9 RTCSCFG_ENABLE_8021Q

默认情况下，RTCS 不发送和接收以太网 802.1Q（VLAN）标签。将该值设为 1，可使 RTCS 发送和接收以太网 802.1Q（VLAN）标签。

IEEE 802.1p 优先级标签由 RTCS_SO_LINK_TX_8021Q_PRIO 和 RTCS_SO_LINK_RX_8021Q_PRIO 套接字选项控制。

VLAN 标识符优先级标签由 RTCS_SO_LINK_TX_8021Q_VID 和 RTCS_SO_LINK_RX_8021Q_VID 套接字选项控制。

8.3.10 RTCSCFG_LINKOPT_8023

默认情况下，RTCS 发送和接收以太网 II 帧。将该值设为 1，可使 RTCS 发送和接收以太网 802.3 和以太网 II 帧。

8.3.11 RTCSCFG_DISCARD_SELF_BCASTS

默认情况下，控制是否丢弃我们发送的所有广播数据包，因为它们很可能是之前集线器的回应。

8.3.12 RTCSCFG_ENABLE_ICMP

默认值为 1。设为 0 来禁止 ICMP 协议。

8.3.13 RTCSCFG_ENABLE_IGMP

默认设为 0。设为 1 以添加对于 IGMP 协议的支持。

8.3.14 RTCSCFG_ENABLE_NAT

默认为 0。设为 1 以添加对 NAT 功能的支持。

8.3.15 RTCSCFG_ENABLE_IPIP

默认值为 0。设为 1 以添加对 IPIP 的支持。

8.3.16 RTCSCFG_ENABLE_RIP

默认值为 0。设为 1 以添加对 RIP 的支持。

8.3.17 RTCSCFG_ENABLE_SNMP

默认值为 0。设为 1 以添加对 SNMP 的支持。

8.3.18 RTCSCFG_ENABLE_SSL

默认值为 0。设为 1 以添加对 SSL/TLS 的支持。

8.3.19 RTCSCFG_ENABLE_IP_REASSEMBLY

默认值为 0。设为 1 以使能 IP 数据包重组。

8.3.20 RTCSCFG_ENABLE_LOOPBACK

默认值设为 0。设为 1 以使能环回接口。

8.3.21 RTCSCFG_ENABLE_UDP

默认值为 1。设为 0 以禁止对 UDP 协议的支持。

8.3.22 RTCSCFG_ENABLE_TCP

默认值为 1。设为 0 以禁止对 TCP 协议的支持。

8.3.23 RTCSCFG_ENABLE_STATS

默认值为 0。设为 1 以添加对网络流量统计数据的支持。

8.3.24 RTCSCFG_ENABLE_GATEWAYS

默认值为 1。设为 0 以禁止对网关的支持。

8.3.25 RTCSCFG_ENABLE_VIRTUAL_ROUTES

默认值为 0。必须为 1 以用于 PPP 或网络隧道。

8.3.26 RTCSCFG_USE_KISS_RNG

默认为 0。必须为 1 以用于 PPP 或网络隧道。

8.3.27 RTCSCFG_ENABLE_ARP_STATS

默认值为 0。设为 1 以使能 ARP 数据包统计数据。

8.3.28 RTCSCFG_PCBS_INIT

PCB（数据包控制块）初始分配数。在应用程序中设置全局变量_RTCSPCB_init 将此值覆盖。

8.3.29 RTCSCFG_LLMNRSRV_PORT

LLMNR 服务器的默认监听端口。根据 RFC 4795，LLMNR 服务器或响应端必须监听 UDP 端口 5355。不建议对其进行修改。

8.3.30 RTCSCFG_LLMNRSRV_HOSTNAME_TTL

默认 TTL 值指示用于 LLMNR 查询器的链接本地主机名有效的时间 (以秒为单位)。默认值为 30 秒（由 RFC4795 推荐)。在高度动态的环境中，如移动 ad-hoc 网络，可能需要降低 TTL 值。

8.3.31 RTCSCFG_PCBS_GROW

PCB（数据包控制块）分配增长间隔。通过设置_RTCSPCB_grow 全局变量在应用程序中改写。

8.3.32 RTCSCFG_PCBS_MAX

PCB（数据包控制块）最大分配数。在应用程序中设置全局变量 RTCSPCB_max 将此值覆盖。

8.3.33 RTCSCFG_MSGPOOL_INIT

RTCS 消息池初始容量。通过设置_RTCS_msgpool_init 变量在应用程序中覆盖。

8.3.34 RTCSCFG_MSGPOOL_GROW

RTCS 消息池增长间隔。通过设置_RTCS_msgpool_grow 变量在应用程序中覆盖。

8.3.35 RTCSCFG_MSGPOOL_MAX

RTCS 消息池最大容量。通过设置_RTCS_msgpool_max 变量在应用程序中覆盖。

8.3.36 RTCSCFG_SOCKET_PART_INIT

RTCS 套接字预分配数。通过设置_RTCS_socket_part_init 在应用程序中覆盖。

8.3.37 RTCSCFG_SOCKET_PART_GROW

RTCS 套接字分配增长间隔。通过设置_RTCS_socket_part_grow 在应用程序中覆盖。

8.3.38 RTCSCFG_SOCKET_PART_MAX

RTCS 套接字最大数。通过设置_RTCS_socket_part_max 在应用程序中覆盖。

8.3.39 RTCSCFG_UDP_RX_BUFFER_SIZE

每个套接字队列数据字节的 UDP 最大值。通过 setsockopt() OPT_RBSIZE 套接字选项在应用程序中覆盖。

8.3.40 RTCSCFG_ENABLE_UDP_STATS

设为 0 用于禁止 UDP 统计数据。

8.3.41 RTCSCFG_ENABLE_TCP_STATS

设为 0 用于禁止 TCP 统计数据。

8.3.42 RTCSCFG_TCP_MAX_CONNECTIONS

默认值为 0。允许同时连接的最大数。定义为 0 表示无限制。

8.3.43 RTCSCFG_TCP_MAX_HALF_OPEN

默认值为 0。允许同时半开放连接的最大数。定义为 0 以禁止 SYN 攻击恢复功能。

8.3.44 RTCSCFG_ENABLE_RIP_STATS

默认值为 RTCSCFG_ENABLE_STATS，使能 RIP 统计数据。

8.3.45 RTCSCFG_QUEUE_BASE

通过设置_RTCSQUEUE_base 在应用程序中覆盖。

8.3.46 RTCSCFG_STACK_SIZE

通过设置_RTCSTASK_stacksize 在应用程序中覆盖。

8.3.47 RTCSCFG_LOG_PCB

默认情况下，RTCS 不在 MQX 内核日志中记录数据包生成和解析。如果应用调用 RTCSLOG_enable()，将该值设为 1 以使 RTCS 记录数据包。

8.3.48 RTCSCFG_LOG_SOCKET_API

默认情况下，无论应用是否调用 RTCSLOG_enable()，RTCS 都不会在 MQX 内核日志中记录套接字 API 调用。将该值设为 1 以使 RTCS 记录套接字 API 调用。

8.3.49 RTCSCFG_ENABLE_IP4

使能 IPv4 协议支持。

默认值为 1。

8.3.50 RTCSCFG_ENABLE_IP6

使能 IPv6 协议支持。

默认值为 0。

8.3.51 RTCSCFG_ND6_NEIGHBOR_CACHE_SIZE

在相邻缓存中的最大条目数（每个接口）。

默认值为 6。

8.3.52 RTCSCFG_ND6_PREFIX_LIST_SIZE

在前缀列表中的最大条目数（每个接口）。

默认值为 4。

8.3.53 RTCSCFG_ND6_ROUTER_LIST_SIZE

在默认路由器列表中的最大条目数（每个接口）。

默认值为 2。

8.3.54 RTCSCFG_IP6_IF_ADDRESSES_MAX

每个通道的 IPv6 地址的最大数。

默认值为 5。

8.3.55 RTCSCFG_IP6_IF_DNS_MAX

可分配给一个接口的 DNSv6 服务器地址的最大数。

默认值为 2。

8.3.56 RTCSCFG_IP6_REASSEMBLY

使能 IPv6 数据包重新汇编。

默认值为 1。

8.3.57 RTCSCFG_IP6_LOOPBACK_MULTICAST

使能 IPv6 组播数据包自身环回。

默认值为 0。

8.3.58 RTCSCFG_ND6_RDNSS

根据 RFC6106，使能递归 DNS 服务器（RDNSS）选项支持。

默认值为 1。

8.3.59 RTCSCFG_ND6_RDNSS_LIST_SIZE

递归 DNS 服务器（RDNSS）地址列表中的最大条目数（每个联网接口）。

RFC6106 指定 RDNSS 地址的条目数为 3。

默认值为 3。

8.3.60 RTCSCFG_ND6_DAD_TRANSMITS

当在临时地址上执行重复地址检测时，发送请求消息的最大数。

默认值为 1。

值 1 为单次传输，后续无重新传输。值 0 为不在临时地址上执行重复地址检测。

8.3.61 RTCSCFG_IP6_MULTICAST_MAX

可在整个系统中同时存在的唯一 IPv6 组播的最大成员数。

默认值为 10。

8.3.62 RTCSCFG_IP6_MULTICAST_SOCKET_MAX

可在一个套接字中同时存在的 IPv6 组播的最大成员数。

默认值为 1。

8.3.63 RTCSCFG_ENABLE_MLD

使能组播监听器发现（MLDv1）协议支持。

默认值为 1。

8.3.64 FTPCCFG_SMALL_FILE_PERFORMANCE_ENANCEMENT

设为 1 - 对于小文件性能更佳 - 小于 4 MB。

8.3.65 FTPCCFG_BUFFER_SIZE

FTP 客户端缓冲区大小。

8.3.66 FTPCCFG_WINDOW_SIZE

FTP 客户端最大 TCP 数据包大小。

8.3.67 RTCSCFG_ECHOSRV_DEBUG_MESSAGES

如果在编译时设为 TRUE，ECHOSRV_task 将向 stderr 打印信息消息。默认值为 TRUE。

8.3.68 RTCSCFG_ECHOSRV_DEFAULT_BUFLEN

用于 ECHOSRV 服务的缓冲区大小。默认值为 1500。

8.3.69 RTCSCFG_ECHOSRV_MAX_TCP_CLIENTS

通过 TCP 协议可以同时连接到 ECHOSRV 服务的客户端的最大数量。默认值为 4。

8.3.70 RTCSCFG_ENABLE_SNMP_STATS

使能 SNMP 统计数据。默认值为 RTCSCFG_ENABLE_STATS。

8.3.71 RTCSCFG_IPCFG_ENABLE_DNS

使能 DNS 域名分析。

8.3.72 RTCSCFG_IPCFG_ENABLE_DHCP

使能 DHCP 绑定（取决于 RTCSCFG_ENABLE_UDP ）。

8.3.73 RTCSCFG_IPCFG_ENABLE_BOOT

使能 TFTP 域名处理和 BOOT 绑定。

8.3.74 ENET 模块硬件加速选项

ENET 模块应用第 3 层的网络加速函数。这些函数设计用于加速各种通用联网协议的进程，如 IP、TCP、UDP 和 ICMP。

8.3.75 BSPCFG_ENET_HW_TX_IP_CHECKSUM_NEW

设为 1 以使能由 ENET 模块产生的 IPv4 头文件校验,用于输出数据包。设为 0 以禁止。

8.3.76 BSPCFG_ENET_HW_TX_PROTOCOL_CHECKSUM_NEW

设为 1 以使能由 ENET 模块产生的 TCP、UDP 和 ICMPv4 校验，用于输出数据包。设为 0 以禁止。

8.3.77 BSPCFG_ENET_HW_RX_IP_CHECKSUM

设为 1 以使能由 ENET 模块验证的 IPv4 头文件校验,用于输入数据包。设为 0 以禁止。

8.3.78 BSPCFG_ENET_HW_RX_PROTOCOL_CHECKSUM_NEW

设为 1 以使能由 ENET 模块验证的 TCP、UDP 和 ICMPv4 校验，用于输入数据包。设为 0 以禁止。

8.3.79 BSPCFG_ENET_HW_RX_MAC_ERR

设为 1 以使能由 ENET 模块丢弃具有 MAC 层（CRC、长度或 PHY）错误的输入帧。设为 0 以禁止。

8.3.80 RTCSCFG_ECHOCLN_DEFAULT_BUFLEN

Echo 客户端应用的缓冲区大小（以字节为单位）。默认为 1500。

8.3.81 RTCSCFG_ECHOCLN_DEFAULT_LOOPCNT

每调用一次 ECHOCLN_process()，Echo 客户端应用循环一次。默认为 1。

8.3.82 RTCSCFG_ECHOCLN_DEBUG_MESSAGES

当为 TRUE 时，Echo 客户端应用向 stderr 打印信息消息。默认为 TRUE。

第 9 章
数据类型

9.1 RTCS 数据类型

RTCS 数据类型	MQX 数据类型	定义位置	注释
_enet_address	**uchar***[6]*	*enet.h*	位于 MQX 源文件中
_enet_handle	**void***	*enet.h*	位于 MQX 源文件中
_ip_address	**uint32_t**	*rtcs.h*	
_ppp_handle	**void***	*ppp.h*	
_task_id	**uint32_t**	*mqx.h*	位于 MQX 源文件中
u_char	**uchar**	*rpctypes.h*	
u_int	**uint32_t**	*rpctypes.h*	
u_long	**uint32_t**	*rpctypes.h*	
u_short	**uint16_t**	*rpctypes.h*	

9.2 按字母顺序排列的 RTCS 数据结构列表

本节提供按字母顺序排列的 RTCS 数据结构列表，并提供以下信息：

- 功能
- 定义
- 字段

9.2.1 addrinfo

该结构由 getaddrinfo()函数使用。

```
typedef struct addrinfo {
        uint16_t        ai_flags;
        uint16_t        ai_family    ;
        uint32_t        ai_socktype;
        uint16_t        ai_protocol;
        unsigned int    ai_addrlen;
        char            *ai_canonname;
        struct sockaddr *ai_addr;
        struct addrinfo *ai_next;
    } addrinfo;
```

ai_flags

由 getaddrinfo()的 hints 参数使用的标志字段应设为 0、AI_CANONNAME、AI_NUMERICHOST 和 AI_PASSIVE 中的一个或多个值的按位或运算值:

- AI_CANONNAME: 如果 AI_CANONNAME 位置位，则成功调用 getaddrinfo() 将返回以 NUL 结尾的字符串, 包含在 addrinfo 结构返回的 ai_canonname 元素中指定主机名的 canonical 名。
- AI_NUMERICHOST: 如果 AI_NUMERICHOST 位置位，这说明应将主机名作为数字字符串对待，用于定义 IPv4 或 IPv6 地址并不应尝试域名解析。
- AI_PASSIVE: 如果 AI_PASSIVE 位置位，这说明返回的套接字地址用于调用 bind()。这种情况下，如果主机名参数为空指针，则套接字地址结构的 IP 地址部分将设为 INADDR_ANY 以用于 IPv4 地址，或设为 IN6ADDR_ANY_INIT 以用于 IPv6 地址。如果 AI_PASSIVE 位未置位，则返回的套接字地址结构将可在调用 connect()中使用，用于以连接为导向的协议，如果选择了无连接协议，该位可用于调用 connect()、sendto()或 sendmsg()。如果主机名为空指针且 AI_PASSIVE 未置位，则套接字地址结构的 IP 地址部分将设为环回地址。

ai_family

协议系列（AF_INET 或 AF_INET6）。

ai_socktype

套接字类型（SOCK_STREAM 或 SOCK_DGRAM）。

ai_protocol

协议（IPPROTO_TCP 或 IPPROTO_UDP）。

ai_addrlen

ai_addr 成员长度。

ai_canonname

主机的规范名称。

ai_addr

套接字地址。

ai_next

指向链表中下一个 *addrinfo* 结构的指针。

9.2.2 ARP_STATS

指向该结构的指针由 ARP_stats() 返回。

```
typedef struct {
  uint32_t             ST_RX_TOTAL;
  uint32_t             ST_RX_MISSED;
  uint32_t             ST_RX_DISCARDED;
  uint32_t             ST_RX_ERRORS;
  uint32_t             ST_TX_TOTAL;
  uint32_t             ST_TX_MISSED;
  uint32_t             ST_TX_DISCARDED;
  uint32_t             ST_TX_ERRORS;
  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;
  uint32_t             ST_RX_REQUESTS;
  uint32_t             ST_RX_REPLIES;
  uint32_t             ST_TX_REQUESTS;
  uint32_t             ST_TX_REPLIES;
  uint32_t             ST_ALLOCS_FAILED;
  uint32_t             ST_CACHE_HITS;
  uint32_t             ST_CACHE_MISSES;
  uint32_t             ST_PKT_DISCARDS;
} ARP_STATS, * ARP_STATS_PTR;
```

ST_RX_TOTAL

已接收（全部）。

ST_RX_MISSED

已接收（由于缺少资源而丢弃）。

ST_RX_DISCARDED

已接收（因所有其他原因而丢弃）。

ST_RX_ERRORS

已接收（具有内部错误）。

ST_TX_TOTAL

已发送（全部）。

ST_TX_MISSED

已发送（由于缺少资源而丢弃）。

ST_TX_DISCARDED

已发送（因所有其他原因而丢弃)。

ST_TX_ERRORS

已发送（具有内部错误)。

ERR_RX

接收错误信息。

ERR_TX

发送错误信息。

ST_RX_REQUESTS

已接收有效 ARP 请求。

ST_RX_REPLIES

已接收有效 ARP 回复。

ST_TX_REQUESTS

已发送 ARP 请求。

ST_TX_REPLIES

已发送 ARP 回复。

ST_ALLOCS_FAILED

ARP_alloc()返回 NULL。

ST_CACHE_HITS

命中 ARP 高速缓存。

ST_CACHE_MISSES

未命中 ARP 高速缓存。

ST_PKT_DISCARDS

由于缺少一个 ARP 条目，数据包丢弃 。

9.2.3 BOOTP_DATA_STRUCT

指向该结构的指针为 RTCS_if_bind_BOOTP() 的输入参数。

```
typedef struct bootp_data_struct
{
  _ip_address  SADDR;
  uchar        SNAME[64];
  uchar        BOOTFILE[128];
  uchar        OPTIONS[64];
} BOOTP_DATA_STRUCT, * BOOTP_DATA_STRUCT_PTR;
```

SADDR

启动文件服务器的 IP 地址。

SNAME

对应于 *SADDR* 的主机名。

BOOTFILE

要加载的启动文件。

OPTIONS

BootP 选项。

9.2.4 DHCP_DATA_STRUCT

指向该结构的指针为 RTCS_if_bind_DHCP() 的参数。

```
typedef struct {
  int32_t    (_CODE_PTR_  CHOICE_FUNC)(uchar *, uint32_t);
  void      (_CODE_PTR_  BIND_FUNC)  (uchar *, uint32_t,
                                      _rtcs_if_handle);
  bool   (_CODE_PTR_  UNBIND_FUNC)(_rtcs_if_handle);
} DHCP_DATA_STRUCT,  *  DHCP_DATA_STRUCT_PTR;
```

CHOICE_FUNC

每次调用时,服务器会接收到一个 DHCP OFFER。如果 *CHOICE_FUNC* 为 0,RTCS 会尝试绑定其接收到的首个提议。

- 第一个参数 — 指向 OFFER 数据包的指针。
- 第二个参数 — OFFER 数据包的长度。

返回 1 以拒收数据包。

返回 0 以接收数据包。

BIND_FUNC

每次调用 DHCP 获取一个租约。如果 *BIND_FUNC* 为 NULL，RTCS 不修改 DHCP 客户端的行为；该函数仅用于通知目的。

- 第一个参数 — 指向 ACK 数据包的指针。
- 第二个参数 — 数据包的长度。
- 第三个参数 — 传递到 *RTCS_if_bind_DHCP()*的句柄。

UNBIND_FUNC

在租约到期且未续约时调用。如果 *UNBIND_FUNC* 为 NULL，则 RTCS 终止 DHCP。

- 参数 — 传递到 RTCS_if_bind_DHCP()的句柄。

返回 TRUE 以尝试获取新租约。

返回 FALSE 以保持接口不绑定。

9.2.5 DHCPSRV_DATA_STRUCT

指向该结构的指针为 DHCPSRV_ippool_add() 的输入参数。

```
typedef struct dhcpsrv_data_struct {
  _ip_address  SERVERID;
  uint32_t      LEASE;
  _ip_address  MASK;
  _ip_address  SADDR;
  uchar        SNAME[64];
  uchar        FILE[128];
} DHCPSRV_DATA_STRUCT, * DHCPSRV_DATA_STRUCT_PTR;
```

SERVERID

服务器的 IP 地址。

LEASE

最大允许租约长度。

MASK

子网掩码。

SADDR

在 DHCP 数据包头文件中的 SADDR 字段。

SNAME

在 DHCP 数据包头文件中的 SNAME 字段。

FILE

在 DHCP 数据包头文件中的 FILE 字段。

9.2.6 DHCPCLN6_STATUS

用于函数 DHCPCLN6_get_status()的返回值列举类型。

```
typedef enum dhcpcln6_status
{ DHCPCLN6_STATUS_BOUND,
    DHCPCLN6_STATUS_UNBOUND,
    DHCPCLN6_STATUS_NOT_RUNNING
}DHCPCLN6_STATUS;
```

DHCPCLN6_STATUS_BOUND - 客户端正在运行，一些地址被 DHCPv6 客户端绑定。

DHCPCLN6_STATUS_UNBOUND - 客户端正在运行, 目前没有地址被 DHCPv6 客户端绑定（消息交换还未完成)。

DHCPCLN6_STATUS_NOT_RUNNING - 客户端未在运行。

9.2.7 DHCPCLN6_PARAM_STRUCT

```
typedef struct dhcpcln6_param_struct
{
    in6_addr                      *preferred;
    _rtcs_if_handle               interface;
    uint32_t                         flags;
    const DHCPCLN6_CALLBACK_TABLE *callbacks;
}DHCPCLN6_PARAM_STRUCT;
```

preferred

用于设备的首选 IPv6 地址。客户端将尝试从服务器获取该地址。

interface

RTCS 句柄将用于 DHCPv6 客户端启动的接口。

flags

用于使能各种特性的客户端标志。

9.2.8 ECHOSRV_PARAM_STRUCT

该结构为用户提供用于 ECHOSRV 服务的配置参数。指向该结构的指针作为输入参数传递到 ECHOSRV_init()。

typedef struct echosrv_param_struct

```
{
  uint16_t                  af;            /* Inet protocol family */
  uint16_t                  port;          /* Listening port */
#if RTCSCFG_ENABLE_IP4
  in_addr                   ipv4_address;  /* Listening IPv4 address */
#endif
#if RTCSCFG_ENABLE_IP6
  in6_addr                  ipv6_address;   /* Listening IPv6 address */
  uint32_t                  ipv6_scope_id;  /* Scope ID for IPv6 */
#endif
  uint32_t                  server_prio;    /* server task priority */
} ECHOSRV_PARAM_STRUCT;
af
AF_INET - to service only IPv4 clients
AF_INET6 - to service only IPv6 clients
AF_INET | AF_INET6 - to service IPv4 or IPv6 clients
```

port

被服务的本地端口号。根据 RFC 862 应使用端口 7。

ipv4_address

监听 IPv4 地址。全零意味着将回复来自任意 IPv4 地址的数据。

ipv6_address

ipv6_scope_id

监听 IPv6 地址。全零意味着来自任意接口的任意 IPv6 地址的数据。

server_prio

ECHOSRV 服务运行于任务中。该参数确定 ECHOSRV 任务的优先级。与 TCP/IP 任务优先级相比，应对其分配一个较低的优先级 (或较高数值)。

9.2.9 ENET_STATS

指向该结构的指针由 ENET_get_stats() 返回。

```
typedef struct {
  uint32_t   ST_RX_TOTAL;
  uint32_t   ST_RX_MISSED;
  uint32_t   ST_RX_DISCARDED;
  uint32_t   ST_RX_ERRORS;
  uint32_t   ST_TX_TOTAL;
  uint32_t   ST_TX_MISSED;
  uint32_t   ST_TX_DISCARDED;
  uint32_t   ST_TX_ERRORS;
  uint32_t   ST_TX_COLLHIST[16];
```

```
  uint32_t   ST_RX_ALIGN;
  uint32_t   ST_RX_FCS;
  uint32_t   ST_RX_RUNT;
  uint32_t   ST_RX_GIANT;
  uint32_t   ST_RX_LATECOLL;
  uint32_t   ST_RX_OVERRUN;
  uint32_t   ST_TX_SQE;
  uint32_t   ST_TX_DEFERRED;
  uint32_t   ST_TX_LATECOLL;
  uint32_t   ST_TX_EXCESSCOLL;
  uint32_t   ST_TX_CARRIER;
  uint32_t   ST_TX_UNDERRUN;
} ENET_STATS, * ENET_STATS_PTR;
```

ST_RX_TOTAL

已接收（全部）。

ST_RX_MISSED

已接收（丢失数据包）。

ST_RX_DISCARDED

已接收（由于未识别协议而丢弃）。

ST_RX_ERRORS

已接收（由于接收错误而丢弃）。

ST_TX_TOTAL

已发送（全部）。

ST_TX_MISSED

已发送（因为发送环已满而丢弃）。

ST_TX_DISCARDED

已发送（因为数据包损坏而丢弃）。

ST_TX_ERRORS

已发送（在传输期间发生错误）。

ST_TX_COLLHIST

已发送（冲突柱状图）。

以下状态用于物理错误或情况。

ST_RX_ALIGN

帧对齐错误。

ST_RX_FCS

CRC 错误。

ST_RX_RUNT

已接收短包。

ST_RX_GIANT

已接收超大包。

ST_RX_LATECOLL

滞后冲突。

ST_RX_OVERRUN

DMA 上溢。

ST_TX_SQE

心跳丢失。

ST_TX_DEFERRED

传输推迟。

ST_TX_LATECOLL

滞后冲突。

ST_TX_EXCESSCOLL

过度冲突。

ST_TX_CARRIER

载体检测丢失。

ST_TX_UNDERRUN

DMA 下溢。

9.2.10 FTPSRV_AUTH_STRUCT

结构定义关于 FTP 服务器用户的验证信息。

```
typedef struct ftpsrv_auth_struct
{
    char* uid;
    char* pass;
    char* path;
}FTPSRV_AUTH_STRUCT;
```

uid

用于表示所用身份的字符串。通常为用户名。

pass

用户密码。

path

在用户登录后设为 FTP 根目录的路径。如果设为 NULL，则使用服务器根目录。

9.2.11 FTPSRV_PARAM_STRUCT

该结构作为 FTPSRV_init()函数的参数使用。

```
typedef struct ftpsrv_param_struct
{
    uint16_t                    af;
    unsigned short              port;
  #if RTCSCFG_ENABLE_IP4
    in_addr                     ipv4_address;
  #endif
  #if RTCSCFG_ENABLE_IP6
    in6_addr                    ipv6_address;
    uint32_t                    ipv6_scope_id;
  #endif
    _mqx_uint                   max_ses;
    bool                        use_nagle;
    uint32_t                    server_prio;
    const char*                 root_dir;
    FTPSRV_AUTH_STRUCT*         auth_table;
} FTPSRV_PARAM_STRUCT;
```

af

服务器使用的地址系列。可能值为：AF_INET（使用 IPv4)、AF_INET6（使用 IPv6)、AF_INET | AF_INET6（同时使用 IPv4 和 IPv6)。

port

用于监听的端口。默认值由 RFC 定义为 21。

ipv4_address

用于监听的 IPv4 地址。只有当使能 IPv4 时，才使用该变量。默认值由宏 FTPSRVCFG_DEF_ADDR 定义。

ipv6_address

用于监听的 IPv6 地址。只有当使能 IPv6 时，才使用该变量。默认值为 in6addr_any。

ipv6_scope_id

IPv6 的范围 ID（接口标识）。默认值为 0。

max_ses

同时连接到服务器的最大用户数。默认值由宏 FTPSRVCFG_DEF_SES_CNT 定义 (2)。

use_nagle

设为 TRUE 以使能用于服务器套接字的 NAGLE 算法。默认为 FALSE - 禁止 NAGLE。

server_prio

服务器任务的优先级。由服务器创建的所有任务，或服务器任务和会话任务均运行于该优先级。

默认值由宏 FTPSRVCFG_DEF_SERVER_PRIO 定义。

root_dir

服务器根目录。FTP 客户端只能访问该目录及其子目录下的文件。

auth_table

用户数组。每个用户都是数组中的一员，最后一个元素必须设为全 NULL，作为收尾。

9.2.12 HTTPSRV_PARAM_STRUCT

该结构作为 HTTPSRV_init() 函数的参数使用。

```
typedef struct httpsrv_param_struct
{
    uint16_t                    af;
    unsigned short              port;
  #if RTCSCFG_ENABLE_IP4
    in_addr                     ipv4_address;
  #endif
  #if RTCSCFG_ENABLE_IP6
    in6_addr                    ipv6_address;
    uint32_t                    ipv6_scope_id;
  #endif
    _mqx_uint                   max_uri;
    _mqx_uint                   max_ses;
    bool                        use_nagle;
    HTTPSRV_CGI_LINK_STRUCT     *cgi_lnk_tbl;
    HTTPSRV_SSI_LINK_STRUCT     *ssi_lnk_tbl;
    HTTPSRV_PLUGIN_LINK_STRUCT  *plugins;
    HTTPSRV_ALIAS               *alias_tbl;
    uint32_t                    server_prio;
    uint32_t                    script_prio;
    uint32_t                    script_stack;
    char*                       root_dir;
    char*                       index_page;
    HTTPSRV_AUTH_REALM_STRUCT   *auth_table;
```

```
    const HTTPSRV_SSL_STRUCT     *ssl_params;
} HTTPSRV_PARAM_STRUCT;
```

af

服务器使用的地址系列。可能值为：AF_INET（使用 IPv4)、AF_INET6（使用 IPv6)、AF_INET | AF_INET6（同时使用 IPv4 和 IPv6)。

port

用于监听的端口。默认值由宏 HTTPSRVCFG_DEF_PORT 定义。

ipv4_address

用于监听的 IPv4 地址。只有当使能 IPv4 时，才使用该变量。默认值由宏 HTTPSRVCFG_DEF_ADDR 定义。

ipv6_address

用于监听的 IPv6 地址。只有当使能 IPv6 时，才使用该变量。默认值为 in6addr_any。

ipv6_scope_id

IPv6 的范围 ID（接口标识)。默认值为 0。

max_uri

由客户端请求的 URI 最大长度（以字节为单位)。当 URL 超出此长度，将向客户端发送代码 414（Request-URI Too Long）的响应。默认值由宏 HTTPSRVCFG_DEF_URL_LEN 定义。

max_ses

服务器创建的最大会话（连接）数。默认值由宏 HTTPSRVCFG_DEF_SES_CNT 定义。

use_nagle

设为 TRUE 以使能用于服务器套接字的 NAGLE 算法。默认为 FALSE - 禁止 NAGLE。

cgi_lnk_tbl

函数名称和函数指针表用作 CGI 回调。默认为空表（NULL 指针)。

ssi_lnk_tbl

函数名称和函数指针表用作 SSI 回调。默认为空表（NULL 指针)。

alias_tbl

目录别名表。关于别名功能的说明请参见别名 章节。

server_prio

服务器任务的优先级。由服务器创建的所有任务（服务器任务和会话任务）均运行于该优先级。默认值由宏 HTTPSRVCFG_DEF_SERVER_PRIO 定义。

script_prio

脚本句柄任务的优先级。该值应比 server_prio 小或与其一致。默认值由宏 HTTPSRVCFG_DEF_SERVER_PRIO 定义。

script_stack

脚本句柄任务的协议栈大小（以字节为单位）。根据 CGI 和 SSI 回调的内存要求设置该变量值。默认值为 750 字节。

root_dir

服务器的根目录。客户端可用的所有文件存放于该变量定义的路径中。默认值为 "tfs:" (根目录设为简单文件系统)。

index_page

当根目录被请求时发送给客户端的默认页。默认值由宏 HTTPSRVCFG_DEF_INDEX_PAGE 定义。

auth_table

验证区域表。默认为空表（NULL 指针）。

plugins

指向服务器插件列表的指针。

ssl_params

指向 HTTPSRV_SSL_STRUCT SSL 参数结构的指针。该变量为可选，并且可设为 NULL。

9.2.13 HTTPSRV_AUTH_USER_STRUCT

定义用户的结构。用于验证目的。

```
typedef struct httpsrv_auth_user_struct
{
    char* user_id;
    char* password;
}HTTPSRV_AUTH_USER_STRUCT;
```

user_id

用户标识符（用户名等）

password

用户密码。

9.2.14 HTTPSRV_AUTH_REALM_STRUCT

定义验证区域的结构。

```
typedef struct httpsrv_auth_realm_struct
{
    char*                      name;
    char*                      path;
    HTTPSRV_AUTH_TYPE          auth_type;
    HTTPSRV_AUTH_USER_STRUCT* users;
} HTTPSRV_AUTH_REALM_STRUCT;
```

name

区域名称。该字符串作为标识符发送到客户端，从而用户可以确定正确的用户名和密码。

path

验证保护的文件或目录的相对路径。

auth_type

验证类型。数值可为 HTTPSRV_AUTH_INVALID、HTTPSRV_AUTH_BASIC 或 HTTPSRV_AUTH_DIGEST。目前服务器（v2.0）只支持基本验证。

users

属于同一个区域内的用户表。

9.2.15 HTTPSRV_CGI_REQ_STRUCT

该结构作为参数传递到用户定义的 CGI 回调函数并包含关于连接、客户端和服务器的基本信息。

```
typedef struct httpsrv_cgi_request_struct
{
    uint32_t             ses_handle;
    HTTPSRV_REQ_METHOD   request_method;
    HTTPSRV_CONTENT_TYPE content_type;
    uint32_t             content_length;
    uint32_t             server_port;
    char*               remote_addr;
    char*               server_name;
    char*               script_name;
    char*               server_protocol;
    char*               server_software;
    char*               query_string;
    char*               gateway_interface;
```

```
    char*                   remote_user;
    HTTPSRV_AUTH_TYPE       auth_type;
}HTTPSRV_CGI_REQ_STRUCT;
```

ses_handle

会话句柄。该值要求作为参数读取和写入服务器（向客户端发送响应)。

request_method

提出请求的客户端使用的方法。通过枚举 HTTPSRV_REQ_METHOD 可对该变量定义任意值。在处理请求前，用户回调必须检查其是否具有正确的类型。

content_type

从客户端发送到服务器的请求实体类型。通过列举 HTTPSRV_CONTENT_TYPE 可对该变量定义任意值。

content_length

请求实体的长度（以字节为单位)。

server_port

建立与客户端连接的本地端口。

remote_addr

远程（客户端）IP 地址。可以为 IPv4 或 IPv6 地址。

server_name

服务器 IP 地址或主机名。可以为 IPv4 或 IPv6 地址。

script_name

调用的 CGI 函数名。这对脚本自识别非常有用。

server_protocol

服务器用来与客户端通信的协议（HTTP/1.0)。

server_software

字符串识别服务器软件的名称和版本。

query_string

问号后的部分请求 URI。

gateway_interface

常用网关接口的类型和版本（CGI/1.1)。

remote_user

客户端发送的用户名作为验证过程的一部分。

auth_type

使用的验证类型。

9.2.16 HTTPSRV_CGI_RES_STRUCT

由用户 CGI 函数生成的响应结构。该结构被要求作为函数 **httpsrv_cgi_write()**的参数。整个结构必须填充用户 CGI 回调。

```
typedef struct httpsrv_cgi_response_struct
{
    uint32_t              ses_handle;
    HTTPSRV_CONTENT_TYPE content_type;
    uint32_t              content_length;
    uint32_t              status_code;
    char*                data;
    uint32_t              data_length;
}HTTPSRV_CGI_RES_STRUCT;
```

ses_handle

用于 CGI 读/写操作的会话的句柄。

content_type

CGI 生成的响应的内容类型。

content_length

CGI 脚本的响应实体长度。

status_code

HTTP 响应状态代码。典型值为 200（响应 OK）或 404（未找到)。

data

指向作为响应客户端写入的用户数据的指针。

data_length

用户数据大小（以字节为单位)。

9.2.17 HTTPSRV_SSI_PARAM_STRUCT

传递到用户 SSI（包括服务器端）回调的参数结构。

```
typedef struct httpsrv_ssi_param_struct
{
    uint32_t ses_handle;
    char*    com_param;
}HTTPSRV_SSI_PARAM_STRUCT;
```

ses_handle

要求用于在 SSI 回调中进行写操作的会话的句柄。

com_param

来自网页（第一个冒号后的所有内容）的 SSI 命令的参数。

9.2.18 HTTPSRV_SSI_LINK_STRUCT

定义 SSI 回调表的行的结构。

```
typedef struct httpsrv_ssi_link_struct
{
    char* fn_name;
    HTTPSRV_SSI_CALLBACK_FN callback;
} HTTPSRV_SSI_LINK_STRUCT;
```

fn_name

函数的名称/标签。例如在解析*.shtm 文件的*.shtlm 时遇到<%usbstat:test%>字符串，则会调用名为“usbstat”的函数，该函数的参数字符串设为“test”。

callback

当在 SSI 文件中找到字符串<%fn_name%>时，指向所调用函数的指针。

stack

SSI 的协议栈大小。如果设为 0，将使用默认的脚本句柄任务。否则，会创建新的独立任务以处理协议栈设为该值的脚本。

9.2.19 HTTPSRV_CGI_LINK_STRUCT

定义 CGI 回调表的行的结构。

```
typedef struct httpsrv_ssi_link_struct
{
    char* fn_name;
    HTTPSRV_SSI_CALLBACK_FN callback;
uint32_t stack;
} HTTPSRV_SSI_LINK_STRUCT;
```

fn_name

函数的名称/标签。假设客户端请求 rtcdata.cgi 文件，将调用具有标签“rtcdata”的函数。

callback

当请求文件名 fn_name.cgi 时，指向该函数的指针。

stack

CGI 的协议栈大小。如果设为 0，将使用默认的脚本句柄任务。否则，会创建新的独立任务以处理协议栈设为该值的脚本。

9.2.20 HTTPSRV_ALIAS

该结构定义服务器别名表中的一个项目。

```
typedef struct httpsrv_alias
{
    char* alias;
    char* path;
}HTTPSRV_ALIAS;
```

alias

用户定义的别名路径名称。当访问文件时，该名称用作 URI 的一部分。

path

要应用别名的文件系统路径。

9.2.21 HTTPSRV_PLUGIN_STRUCT

定义网络服务器插件的结构:

```
typedef struct httpsrv_plugin_struct
{
    HTTPSRV_PLUGIN_TYPE type;
    void                *data;
}HTTPSRV_PLUGIN_STRUCT;
```

type

插件类型。只支持 HTTPSRV_WS_PLUGIN。

data

指向插件数据的指针。

9.2.22 HTTPSRV_PLUGIN_LINK_STRUCT

用于链接资源（URI）到服务器插件的结构。

```
typedef struct httpsrv_plugin_link_struct
{
    char                  *resource;
    HTTPSRV_PLUGIN_STRUCT *plugin;
}HTTPSRV_PLUGIN_LINK_STRUCT;
```

resource

导致插件调用的资源路径（相对于服务器根目录)。

plugin

指向插件结构的指针。

9.2.23 HTTPSRV_SSL_STRUCT

SSL 参数结构，在 HTTP 服务器初始化 HTTPS 期间使用。

```
typedef struct httpsrv_ssl_struct
{
    char*              cert_file;
    char*              priv_key_file;
}HTTPSRV_SSL_STRUCT;
```

cert_file

HTTPS 服务器认证文件的路径。

priv_key_file

HTTPS 服务器私钥文件的路径。

9.2.24 PING_PARAM_STRUCT

```
typedef struct ping_param_struct
{
    sockaddr        addr;
    uint32_t             timeout;
    uint16_t             id;
    uint8_t              hop_limit;
    void          *data_buffer;
    uint32_t             data_buffer_size;
    uint32_t             round_trip_time;
}PING_PARAM_STRUCT, * PING_PARAM_STRUCT_PTR;
```

9.2.25 ICMP_STATS

指向该结构的指针由 ICMP_stats() 返回。

```
typedef struct {
  uint32_t             ST_RX_TOTAL;
  uint32_t             ST_RX_MISSED;
  uint32_t             ST_RX_DISCARDED;
  uint32_t             ST_RX_ERRORS;
  uint32_t             ST_TX_TOTAL;
  uint32_t             ST_TX_MISSED;
  uint32_t             ST_TX_DISCARDED;
  uint32_t             ST_TX_ERRORS;
  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;
  uint32_t             ST_RX_BAD_CODE;
  uint32_t             ST_RX_BAD_CHECKSUM;
  uint32_t             ST_RX_SMALL_DGRAM;
  uint32_t             ST_RX_RD_NOTGATE;
  uint32_t             ST_RX_DESTUNREACH;
  uint32_t             ST_RX_TIMEEXCEED;
  uint32_t             ST_RX_PARMPROB;
  uint32_t             ST_RX_SRCQUENCH;
  uint32_t             ST_RX_REDIRECT;
  uint32_t             ST_RX_ECHO_REQ;
  uint32_t             ST_RX_ECHO_REPLY;
  uint32_t             ST_RX_TIME_REQ;
  uint32_t             ST_RX_TIME_REPLY;
  uint32_t             ST_RX_INFO_REQ;
  uint32_t             ST_RX_INFO_REPLY;
  uint32_t             ST_RX_OTHER;
  uint32_t             ST_TX_DESTUNREACH;
  uint32_t             ST_TX_TIMEEXCEED;
  uint32_t             ST_TX_PARMPROB;
  uint32_t             ST_TX_SRCQUENCH;
  uint32_t             ST_TX_REDIRECT;
  uint32_t             ST_TX_ECHO_REQ;
  uint32_t             ST_TX_ECHO_REPLY;
  uint32_t             ST_TX_TIME_REQ;
  uint32_t             ST_TX_TIME_REPLY;
  uint32_t             ST_TX_INFO_REQ;
  uint32_t             ST_TX_INFO_REPLY;
  uint32_t             ST_TX_OTHER;
} ICMP_STATS, * ICMP_STATS_PTR;
```

9.2.25.1 ST_RX_TOTAL

已接收的数据包总数。

ST_RX_MISSED

由于缺少资源而丢弃的输入数据包。

ST_RX_DISCARDED

由于所有其他原因而丢弃的输入数据包。

ST_RX_ERRORS

在处理输入数据包时检测到内部错误。

ST_TX_TOTAL

已发送的数据包总数。

ST_TX_MISSED

由于缺少资源而丢弃的发送数据包。

ST_TX_DISCARDED

由于所有其他原因而丢弃的发送数据包。

ST_TX_ERRORS

在尝试发送数据包时检测到内部错误。

ERR_RX

接收错误信息。

ERR_TX

发送错误信息。

以下内容包含在 *ST_RX_DISCARDED* 中:

ST_RX_BAD_CODE

具有未识别代码的数据报。

ST_RX_BAD_CHECKSUM

具有无效校验和的数据报。

ST_RX_SMALL_DGRAM

数据报小于头文件。

ST_RX_RD_NOTGATE

来自非网关的重定向接收。

每个 *ICMP* 类型的状态。

ST_RX_DESTUNREACH

接收目的地不可到达。

ST_RX_TIMEEXCEED

接收超时。

ST_RX_PARMPROB

接收参数问题。

ST_RX_SRCQUENCH

接收源终止。

ST_RX_REDIRECT

接收重定向。

ST_RX_ECHO_REQ

接收回送请求。

ST_RX_ECHO_REPLY

接收回送答复。

ST_RX_TIME_REQ

接收时间戳请求。

ST_RX_TIME_REPLY

接收时间戳回复。

ST_RX_INFO_REQ

接收信息请求。

ST_RX_INFO_REPLY

接收信息回复。

ST_RX_OTHER

接收所有其他类型。

ST_TX_DESTUNREACH

发送目的地不可到达。

ST_TX_TIMEEXCEED

发送超时。

ST_TX_PARMPROB

发送参数问题。

ST_TX_SRCQUENCH

发送源终止。

ST_TX_REDIRECT

发送重定向。

ST_TX_ECHO_REQ

发送回送请求。

ST_TX_ECHO_REPLY

发送回送答复。

ST_TX_TIME_REQ

发送时间戳请求。

ST_TX_TIME_REPLY

发送时间戳答复。

ST_TX_INFO_REQ

发送信息请求。

ST_TX_INFO_REPLY

发送信息答复。

ST_TX_OTHER

发送所有其他类型。

9.2.26 IGMP_STATS

指向该结构的指针由 IGMP_stats() 返回。

```
typedef struct {
   uint32_t  ST_RX_TOTAL;
   uint32_t  ST_RX_MISSED;
   uint32_t  ST_RX_DISCARDED;
   uint32_t  ST_RX_ERRORS;
   uint32_t  ST_TX_TOTAL;
   uint32_t  ST_TX_MISSED;
   uint32_t  ST_TX_DISCARDED;
   uint32_t  ST_TX_ERRORS;
   RTCS_ERROR_STRUCT ERR_RX;
   RTCS_ERROR_STRUCT ERR_TX;
   uint32_t  ST_RX_BAD_TYPE;
   uint32_t  ST_RX_BAD_CHECKSUM;
   uint32_t  ST_RX_SMALL_DGRAM;
   uint32_t  ST_RX_QUERY;
   uint32_t  ST_RX_REPORT;
   uint32_t  ST_TX_QUERY;
   uint32_t  ST_TX_REPORT;
} IGMP_STATS, * IGMP_STATS_PTR;
```

ST_RX_BAD_TYPE

具有未识别代码的数据报。

ST_RX_BAD_CHECKSUM

具有无效校验和的数据报。

ST_RX_SMALL_DGRAM

数据报小于头文件。

ST_RX_QUERY

接收询问。

ST_RX_REPORT

接收报告。

ST_TX_QUERY

发送查询。

ST_TX_REPORT

发送报告。

9.2.27 in_addr

在以下结构中的地址字段结构:

- *ip_mreq*
- *sockaddr_in*

```
typedef struct in_addr {
   _ip_address s_addr;
} in_addr;
```

s_addr

IP 地址。

9.2.28 in6_addr

在这些结构中作为 IPv6 地址字段使用:

- *ipv6_mreq*
- *sockaddr_in6*

```
typedef struct in6_addr
{
    union
    {
        uint8_t     _u6_addr8[16];
        uint16_t    _u6_addr16[8];
        uint32_t            _u6_addr32[4];
    } _u6_addr;
}in6_addr;
#define s6_addr   _u6_addr._u6_addr8
```

s6_addr

128 位 IPv6 地址。

9.2.29 ip_mreq

IPv4 组播组。

```
typedef struct ip_mreq {
  in_addr  imr_multiaddr;
  in_addr  imr_interface;
} ip_mreq;
```

imr_multiaddr

组播 IPv4 地址。

imr_interface

本地 IP 地址。

9.2.30 ipv6_mreq

IPv6 组播组。

```
typedef struct ipv6_mreq
{
    in6_addr      ipv6imr_multiaddr;
    unsigned int  ipv6imr_interface;
} ipv6_mreq;
```

ipv6imr_multiaddr

组的 IPv6 组播地址。

ipv6imr_interface

接口索引。这与范围区索引相同，用于定义网络接口。

9.2.31 IP_STATS

指向该结构的指针由 inet_pton() 返回。

```
typedef struct {
  uint32_t            ST_RX_TOTAL;
  uint32_t            ST_RX_MISSED;
  uint32_t            ST_RX_DISCARDED;
  uint32_t            ST_RX_ERRORS;
  uint32_t            ST_TX_TOTAL;
  uint32_t            ST_TX_MISSED;
  uint32_t            ST_TX_DISCARDED;
  uint32_t            ST_TX_ERRORS;
  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;
  uint32_t            ST_RX_HDR_ERRORS;
  uint32_t            ST_RX_ADDR_ERRORS;
  uint32_t            ST_RX_NO_PROTO;
  uint32_t            ST_RX_DELIVERED;
  uint32_t            ST_RX_FORWARDED;
  uint32_t            ST_RX_BAD_VERSION;
  uint32_t            ST_RX_BAD_CHECKSUM;
  uint32_t            ST_RX_BAD_SOURCE;
  uint32_t            ST_RX_SMALL_HDR;
  uint32_t            ST_RX_SMALL_DGRAM;
  uint32_t            ST_RX_SMALL_PKT;
  uint32_t            ST_RX_TTL_EXCEEDED;
  uint32_t            ST_RX_FRAG_RECVD;
  uint32_t            ST_RX_FRAG_REASMD;
  uint32_t            ST_RX_FRAG_DISCARDED;
  uint32_t            ST_TX_FRAG_SENT;
  uint32_t            ST_TX_FRAG_FRAGD;
  uint32_t            ST_TX_FRAG_DISCARDED
} IP_STATS, * IP_STATS_PTR;
```

ST_RX_TOTAL

已接收的数据包总数。

ST_RX_MISSED

由于缺少资源而丢弃的输入数据包。

ST_RX_DISCARDED

由于所有其他原因而丢弃的输入数据包。

ST_RX_ERRORS

在处理输入数据包时检测到内部错误。

ST_TX_TOTAL

已发送的数据包总数。

ST_TX_MISSED

由于缺少资源而丢弃的发送数据包。

ST_TX_DISCARDED

由于所有其他原因而丢弃的发送数据包。

ST_TX_ERRORS

在尝试发送数据包时检测到内部错误。

ERR_RX

接收错误信息。

ERR_TX

发送错误信息。

ST_RX_HDR_ERRORS

丢弃（IP 头文件错误)。

ST_RX_ADDR_ERRORS

丢弃（非法目的地)。

ST_RX_NO_PROTO

数据报大于帧。

ST_RX_DELIVERED

数据报发送至较高层。

ST_RX_FORWARDED

数据报转发。

以下内容包含在 *ST_RX_DISCARDED* 和 *ST_RX_HDR_ERRORS* 中。

ST_RX_BAD_VERSION

数据报版本不为 4。

ST_RX_BAD_CHECKSUM

具有无效校验和的数据报。

ST_RX_BAD_SOURCE

数据报具有无效源地址。

ST_RX_SMALL_HDR

数据报的头文件太小。

ST_RX_SMALL_DGRAM

数据报小于头文件。

ST_RX_SMALL_PKT

数据报大于帧。

ST_RX_TTL_EXCEEDED

数据报路由的 TTL = 0。

ST_RX_FRAG_RECVD

接收 IP 片段。

ST_RX_FRAG_REASMD

重汇编数据报。

ST_RX_FRAG_DISCARDED

丢弃代码片段。

ST_TX_FRAG_SENT

发送代码片段。

ST_TX_FRAG_FRAGD

片段式数据报。

ST_TX_FRAG_DISCARDED

分段失败。

9.2.32 IPCFG_IP_ADDRESS_DATA

接口地址结构。

```
typedef uint32_t _ip_address;
typedef struct ipcfg_ip_address_data
{
    _ip_address    ip;
    _ip_address mask;
    _ip_address router;
} IPCFG_IP_ADDRESS_DATA, * IPCFG_IP_ADDRESS_DATA_PTR;
```

ip

IP 地址

mask

掩码

route

网关

9.2.33 IPCP_DATA_STRUCT

指向该结构的指针为 RTCS_if_bind_IPCP() 的参数。

```
typedef struct {
  void (_CODE_PTR_  IP_UP) (void*);
  void (_CODE_PTR_  IP_DOWN) (void*);
  void             *IP_PARAM;
  unsigned          ACCEPT_LOCAL_ADDR  : 1;
  unsigned          ACCEPT_REMOTE_ADDR : 1;
  unsigned          DEFAULT_NETMASK    : 1;
  unsigned          DEFAULT_ROUTE      : 1;
  unsigned          NEG_LOCAL_DNS      : 1;
  unsigned          NEG_REMOTE_DNS     : 1;
  unsigned          ACCEPT_LOCAL_DNS   : 1;
                      /*Ignored if NEG_LOCAL_DNS = 0. */
  unsigned          ACCEPT_REMOTE_DNS  : 1;
                      /*Ignored if NEG_REMOTE_DNS = 0. */
  unsigned                             : 0;

  _ip_address       LOCAL_ADDR;
  _ip_address       REMOTE_ADDR;
  _ip_address       NETMASK;
                      /* Ignored if DEFAULT_NETMASK = 1. */
  _ip_address       LOCAL_DNS;
                      /* Ignored if NEG_LOCAL_DNS = 0. */
  _ip_address       REMOTE_DNS;
                      /* Ignored if NEG_REMOTE_DNS = 0. */
  } IPCP_DATA_STRUCT, * IPCP_DATA_STRUCT_PTR;
```

IP_UP

IP_DOWN

IP_PARAM

RTCS 调用	使用	当 IPCP 成功时
IP_UP	*IP_PARAM*	进入打开状态。
IP_DOWN	*IP_PARAM*	离开打开状态。

ACCEPT_LOCAL_ADDR

LOCAL_ADDR

IPCP 尝试协商 *LOCAL_ADDR* 作为其本地 IP 地址。

如果 ACCEPT_LOCAL_ADDR 为:	IPCP 操作
TRUE	允许对等端协商一个不同的本地 IP 地址。
FALSE	仅接受 *LOCAL_ADDR* 作为其本地 IP 地址。

ACCEPT_REMOTE_ADDR

REMOTE_ADDR

IPCP 尝试协商 *REMOTE_ADDR* 作为其对等端 IP 地址。

如果 ACCEPT_REMOTE_ADDR 为:	IPCP 操作
TRUE	允许对等端协商一个不同的对等端 IP 地址。
FALSE	仅接受 *REMOTE_ADDR* 作为其对等端 IP 地址。

NETMASK

DEFAULT_NETMASK

如果 DEFAULT_NETMASK 为:	IPCP 操作
TRUE	基于协商的本地和对等端 IP 地址，动态计算链路的网络掩码。
FALSE	IPCP 总是使用 *NETMASK* 作为网络掩码。

DEFAULT_ROUTE

如果 *DEFAULT_ROUTE* 为 TRUE，IPCP 安装对等端作为 IP 路由表中的默认网关。

ACCEPT_LOCAL_DNS

NEG_LOCAL_DNS

LOCAL_DNS

控制 RTCS 是否协商 DNS 服务器地址由本地分解器使用。如果 *ACCEPT_LOCAL_DNS* 为 TRUE，则对等端可以覆盖 *LOCAL_DNS*。

如果 NEG_LOCAL_DNS 为:	IPCP 操作
TRUE	尝试协商 *LOCAL_DNS* 作为 DNS 服务器地址，该地址由本地分解器使用。
FALSE	不尝试协商用于本地分解器的 DNS 服务器地址。

ACCEPT_REMOTE_DNS

NEG_REMOTE_DNS

REMOTE_DNS

控制 RTCS 是否协商 DNS 服务器地址对等端分解器使用。如果 *ACCEPT_REMOTE_DNS* 为 TRUE，则对等端可以覆盖 *REMOTE_DNS*。

如果 NEG_REMOTE_DNS 为	IPCP 操作
TRUE	尝试协商 REMOTE_DNS 作为 DNS 服务器地址，该地址由对等端分解器使用。
FALSE	不尝试协商用于对等端分解器的 DNS 服务器地址。

9.2.34 IPIF_STATS

指向该结构的指针由 IPIF_stats() 返回。

```
typedef struct {
  uint32_t            ST_RX_TOTAL;
  uint32_t            ST_RX_MISSED;
  uint32_t            ST_RX_DISCARDED;
  uint32_t            ST_RX_ERRORS;
  uint32_t            ST_TX_TOTAL;
  uint32_t            ST_TX_MISSED;
  uint32_t            ST_TX_DISCARDED;
  uint32_t            ST_TX_ERRORS;
  RTCS_ERROR_STRUCT  ERR_RX;
  RTCS_ERROR_STRUCT  ERR_TX;
  uint32_t            ST_RX_OCTETS;
  uint32_t            ST_RX_UNICAST;
  uint32_t            ST_RX_MULTICAST;
  uint32_t            ST_RX_BROADCAST;
  uint32_t            ST_TX_OCTETS;
  uint32_t            ST_TX_UNICAST;
  uint32_t            ST_TX_MULTICAST;
  uint32_t            ST_TX_BROADCAST;
} IPIF_STATS, * IPIF_STATS_PTR;
```

ST_RX_TOTAL

已接收的数据包总数。

ST_RX_MISSED

由于缺少资源而丢弃的输入数据包。

ST_RX_DISCARDED

由于所有其他原因而丢弃的输入数据包。

ST_RX_ERRORS

在处理输入数据包时检测到内部错误。

ST_TX_TOTAL

已发送的数据包总数。

ST_TX_MISSED

由于缺少资源而丢弃的发送数据包。

ST_TX_DISCARDED

由于所有其他原因而丢弃的发送数据包。

ST_TX_ERRORS

在尝试发送数据包时检测到内部错误。

ERR_RX

接收错误信息。

ERR_TX

发送错误信息。

ST_RX_OCTETS

已接收的总字节数。

ST_RX_UNICAST

已接收单播数据包。

ST_RX_MULTICAST

已接收组播数据包。

ST_RX_BROADCAST

已接收广播数据包。

ST_TX_OCTETS

已发送的总字节数。

ST_TX_UNICAST

已发送单播数据包。

ST_TX_MULTICAST

已发送组播数据包。

ST_TX_BROADCAST

已发送广播数据包。

9.2.35 LLMNRSRV_PARAM_STRUCT

```
typedef struct llmnrsrv_param_struct
{
    _rtcs_if_handle                 interface;
    LLMNRSRV_HOST_NAME_STRUCT       *host_name_table;
    uint16_t                        af;
    uint32_t                        task_prio;
} LLMNRSRV_PARAM_STRUCT;
```

Interface

LLMNR 服务器正在监听的接口的 RTCS 句柄。

host_name_table

指向主机名表的指针。表的最后一个条目必须为零。

af

服务器使用的地址系列。可能值为：AF_INET（使用 IPv4）、AF_INET6（使用 IPv6）、AF_INET | AF_INET6（使用 IPv4 和 IPv6）。该参数为可选。默认情况下，服务器将使用所有使能的地址系列。

task_prio

该参数确定 LLMNRSRV 任务的优先级。与 TCP/IP 任务优先级相比，应对其分配一个较低的优先级（或较高数值）。该参数为可选。默认情况下，优先级设为由 RTCSCFG_LLMNRSRV_SERVER_PRIO 定义的值。

9.2.36 LLMNRSRV_HOST_NAME_STRUCT

定义 LLMNR 主机名表的行的结构。表的最后一行必须为零以标记这是表的结束位置。

```
typedef struct llmnrsrv_host_name_struct
{
    char                        *host_name;
    uint32_t                    host_name_ttl;
} LLMNRSRV_HOST_NAME_STRUCT;
```

host_name

链路本地主机名由 LLMNR 服务器通知（以 null 结束）。

host_name_ttl

TTL 值指示用于 LLMNR 查询器的链路本地主机名有效的秒数（可选）。该参数为可选。默认值由 RTCSCFG_LLMNRSRV_HOSTNAME_TTL 定义。

9.2.37 nat_ports

Freescale MQX NAT 用来控制端口之间的范围，以及指定最小和最大端口号。

```
typedef struct {
   uint16_t  port_min;
   uint16_t  port_max;
} nat_ports;
```

PORT_MIN

最小端口号。

PORT_MAX

最大端口号。

9.2.38 NAT_STATS

网络地址转换统计数据。

```
typedef struct {
   uint32_t  ST_SESSIONS;
   uint32_t  ST_SESSIONS_OPEN;
   uint32_t  ST_SESSIONS_OPEN_MAX;
   uint32_t  ST_PACKETS_TOTAL;
   uint32_t  ST_PACKETS_BYPASS;
   uint32_t  ST_PACKETS_PUB_PRV;
   uint32_t  ST_PACKETS_PUB_PRV_ERR;
   uint32_t  ST_PACKETS_PRV_PUB;
   uint32_t  ST_PACKETS_PRV_PUB_ERR;
} NAT_STATS, * NAT_STATS_PTR;
```

ST_SESSIONS

到目前为止创建的会话总数。

ST_SESSIONS_OPEN

当前打开的会话数。

ST_SESSIONS_OPEN_MAX

到目前为止同时打开的最大会话数。

ST_PACKETS_TOTAL

由 Freescale MQX NAT 处理的数据包数量。

ST_PACKETS_BYPASS

未修改的数据包数量。

ST_PACKETS_PUB_PRV

从公共到专用区域的数据包数量。

ST_PACKETS_PUB_PRV_ERR

从公共到专用区域的具有错误的数据包数量 (丢弃具有错误的数据包)。

ST_PACKETS_PRV_PUB

从专用到公共区域的数据包数量。

ST_PACKETS_PRV_PUB_ERR

从专用到公共区域的具有错误的数据包数量 (丢弃具有错误的数据包)。

9.2.39 nat_timeouts

由 Freescale MQX NAT 用来确定非活动超时设置。

```
typedef struct {
   uint32_t  timeout_tcp;
   uint32_t  timeout_fin;
   uint32_t  timeout_udp;
} nat_timeouts;
```

TIMEOUT_TCP

非活动超时设置用于 TCP 会话。

TIMEOUT_FIN

非活动超时设置用于 TCP 会话，其中 FIN 或 RST 位置位。

TIMEOUT_UDP

非活动超时设置用于 UDP 或 ICMP 会话。

9.2.40 PPP_PARAM_STRUCT

用作 PPP 设备初始化的参数。

typedef struct ppp_param_struct

```
{
    char*                      device;
    void (_CODE_PTR_ up)       (void *);
    void (_CODE_PTR_ down)     (void *);
    void                       *callback_param;
    _rtcs_if_handle            if_handle;
    _ip_address                local_addr;
    _ip_address                remote_addr;
```

```
    int                    listen_flag;
}PPP_PARAM_STRUCT;
```

device

低层通信设备名。所有 PPP 数据通过该设备发送。

up

当 PPP 链路上行时调用的函数。

down

当 PPP 链路下行时调用的函数。

callback_param

用于 UP/DOWN 回调的函数。

if_handle

ipcp 接口的句柄。PPP 存放该参数的 IPCP 设备句柄。这可用来读取 PPP 链路的远程和本地 IP 地址。

local_addr

要在 PPP 上使用的本地 IP 地址。只有当 listen_flag 设为 TRUE 时才相关。

remote_addr

设置为远程对等端的 IP 地址。只有当 listen_flag 设为 TRUE 时才相关。

listen_flag

用来确定 PPP 应以监听模式（TRUE）还是连接模式（FALSE）开始的标志。

9.2.41 PPP_SECRET

由 PPP 驱动程序用于 PAP 和 CHAP 对等端验证。

```
typedef struct {
  uint16_t    PPP_ID_LENGTH;
  uint16_t    PPP_PW_LENGTH;
  char   *PPP_ID_PTR;
  char   *PPP_PW_PTR;
} PPP_SECRET, * PPP_SECRET_PTR;
```

PPP_ID_LENGTH

PPP_ID_PTR 数组中的字节数。

PPP_PW_LENGTH

PPP_PW_PTR 数组中的字节数。

PPP_ID_PTR

指向表示远程实体 ID（如主机名或用户 ID）数组的指针。

PPP_PW_PTR

指向表示与远程实体 ID 相关的密码数组的指针。

9.2.42 RTCS_ERROR_STRUCT

协议错误统计数据。该结构包括位于以下统计数据结构中的字段 *ERR_TX* 和 *ERR_RX*:

- *ARP_STATS*
- *ICMP_STATS*
- *IGMP_STATS*
- *IP_STATS*
- *IPIF_STATS*
- *TCP_STATS*
- *UDP_STATS*

```
typedef struct {
  uint32_t    ERROR;
  uint32_t    PARM;
  _task_id   TASK_ID;
  uint32_t    TASKCODE;
  void     *MEMPTR;
  bool     STACK;
} RTCS_ERROR_STRUCT, * RTCS_ERROR_STRUCT_PTR;
```

ERROR

说明协议错误的代码。

PARM

与协议错误相关的参数。

TASK_ID

设置代码的任务的任务 ID。

TASKCODE

设置代码的任务的任务错误代码。

MEMPTR

MQX RTOS 分配的最高内核-存储器地址。

STACK

用于设置代码的任务的协议栈是否超出其限制范围。

9.2.43 RTCS_IF_STRUCT

用于设备接口的回调函数。指向该结构的指针为 RTCS_if_add() 的参数。要使用接口的默认表，则需要使用下表中定义的常数。

接口	RTCS_if_add()的参数
以太网	RTCS_IF_ENET
本地环回	RTCS_IF_LOCALHOST
PPP	RTCS_IF_PPP

```
typedef struct {
   uint32_t (_CODE_PTR_  OPEN) (struct ip_if *);
   uint32_t (_CODE_PTR_  CLOSE)(struct ip_if *);
   uint32_t (_CODE_PTR_  SEND) (struct ip_if *,
                               struct rtcspcb *,
                               _ip_address,
                               _ip_address);
   uint32_t (_CODE_PTR_  JOIN) (struct ip_if *,
                               _ip_address);
   uint32_t (_CODE_PTR_  LEAVE)(struct ip_if *,
                               _ip_address);
} RTCS_IF_STRUCT, * RTCS_IF_STRUCT_PTR;
```

IP 接口结构（*ip_if*）包含支持 RTCS 发送数据包（以太网）或数据报（PPP）的信息。

OPEN

由 RTCS 调用来注册数据包驱动程序（以太网）或打开链接（PPP）。

- 参数 — 指向 IP 接口结构的指针。

返回状态代码。

CLOSE

由 RTCS 调用来取消注册数据包驱动程序（以太网）或关闭链接（PPP）。

- 参数 — 指向 IP 接口结构的指针。

返回状态代码。

SEND

由 RTCS 调用来发送数据包（以太网）或数据报（PPP)。

- 第一个参数 — 指向 IP 接口结构的指针。
- 第二个参数 — 指向要发送的数据包（以太网）或数据报（PPP）的指针。
- 第三个参数:
  - 针对以太网：要使用的协议（*ENETPROT_IP* 或 *ENETPROT_ARP*)。
  - 针对 PPP：下一跳的源地址。
- 第四个参数:
  - 针对以太网：目的地的 IP 地址。
  - 针对 PPP：下一跳的目标地址。

返回状态代码。

JOIN

由 RTCS 调用来加入组播组（不用于 PPP 接口)。

- 第一个参数 — 指向 IP 接口结构的指针。
- 第二个参数 — 组播组的 IP 地址。

返回状态代码。

LEAVE

由 RTCS 调用来脱离组播组（不用于 PPP 接口)。

- 第一个参数 — 指向 IP 接口结构的指针。
- 第二个参数 — 组播组的 IP 地址。

返回状态代码。

9.2.44 rtcs_fd_set

该结构保存指向套接字描述符的指针。

```
typedef struct tag_rtcs_fd_set
{
```

```
  uint32_t   fd_count;
  uint32_t   fd_array[RTCSCFG_FD_SETSIZE];
} rtcs_fd_set;
```

fd_count

套接字描述符的数量随 RTCS_FD_SET()递增，随 RTCS_FD_CLR()递减。

fd_array

指向套接字描述符的指针作为无符号 32 位整数值保存。

9.2.45 RTCS_protocol_table

以 NULL 结尾的表，定义了当 RTCS 创建时，RTCS 初始化和启动的协议。RTCS 以表中的顺序初始化这些协议。应用只能使用表中的协议。如果您从表中删除一个协议，RTCS 不会将相关代码链接到您的应用，该操作可以降低代码长度。

```
extern uint32_t (_CODE_PTR_ RTCS_protocol_table[])(void);
```

支持的协议

RTCSPROT_IGMP

互联网组管理协议 — 用于组播。

RTCSPROT_UDP

用户数据报协议 — 无连接数据报服务。

RTCSPROT_TCP

传输控制协议 — 可靠的以连接为导向的数据流服务。

RTCSPROT_RIP

路由信息协议 — 需要 UDP。

默认 **RTCS** 协议表

您可以定义您自己的协议表或使用以下 RTCS 提供的默认协议表（位于 *if \rtcsinit.c*）:

```
uint32_t (_CODE_PTR_ RTCS_protocol_table[])(void) = {
   RTCSPROT_IGMP,
   RTCSPROT_UDP,
   RTCSPROT_TCP,
   RTCSPROT_IPIP,
   NULL
```

9.2.46 RTCS_SSL_PARAMS_STRUCT

函数 RTCS_ssl_init()的初始化参数。

```
typedef struct rtcs_ssl_params_struct
{
    char*               cert_file;
    char*               priv_key_file;
    char*               ca_file;
    RTCS_SSL_INIT_TYPE init_type;
}RTCS_SSL_PARAMS_STRUCT;
```

cert_file

应用程序认证文件的路径。

priv_key_file

应用程序私钥文件的路径。

ca_file

CA（认证授权）认证文件的路径。

init_type

初始化类型。可以为 RTCS_SSL_SERVER 或 RTCS_SSL_CLIENT 的值。

RTCS_SSL_SERVER 意味着将初始化用于服务器的 SSL 上下文，
RTCS_SSL_CLIENT

创建用于客户端的 SSL 上下文。

9.2.47 RTCS_TASK

定义 Telnet Server shell 任务。

```
typedef struct {
  char            *NAME;
  uint32_t      PRIORITY;
  uint32_t     STACKSIZE;
  void (_CODE_PTR_    START)(void*);
  void        *ARG;
} RTCS_TASK, * RTCS_TASK_PTR;
```

NAME

任务名称。

PRIORITY

任务优先级。

STACKSIZE

任务协议栈大小。

START

任务入口点。

ARG

任务参数。

9.2.48 RTCS6_IF_ADDR_INFO

```
typedef struct rtcs6_if_addr_info
{
    in6_addr              ip_addr;
    rtcs6_if_addr_state ip_addr_state;
    rtcs6_if_addr_type ip_addr_type;
} RTCS6_IF_ADDR_INFO, * RTCS6_IF_ADDR_INFO_PTR;
```

ip_addr

IPv6 地址。

ip_addr_state

IPv6 地址状态（重复或首选）。

ip_addr_type

IPv6 地址类型（手动设置或使用自动配置）。

9.2.49 ssRTCS6_IF_PREFIX_LIST_ENTRY

前缀列表条目，由 RTCS6_if_get_prefix_list_entry()返回。

```
typedef struct rtc6_if_prefix_list_entry
{
    in6_addr            prefix;
    uint32_t            prefix_length;
} RTCS6_IF_PREFIX_LIST_ENTRY, *RTCS6_IF_PREFIX_LIST_ENTRY_PTR;
```

prefix

IPv6 前缀。

prefix_length

IPv6 前缀长度（以位为单位）。有效的前缀位数。

9.2.50 RTCS6_IF_NEIGHBOR_CACHE_ENTRY

相邻缓冲入口，由 *RTCS6_if_get_neighbor_cache_entry()*返回。

```
typedef struct rtc6_if_neighbor_cache_entry
{
    in6_addr        ip_addr;
    ll_addr_t       ll_addr;
    uint32_t        ll_addr_size;
    bool            is_router;
} RTCS6_IF_NEIGHBOR_CACHE_ENTRY, *RTCS6_IF_NEIGHBOR_CACHE_ENTRY_PTR;
```

ip_addr

相邻的链路单播 IPv6 地址。

ll_addr

链路层地址。实际大小由 *ll_addr_size* 定义。

ll_addr_size

链路层地址大小。

is_router

标志用于指示相邻的是路由器（*TRUE*）还是主机（*FALSE*）。

9.2.51 rtcs6_if_addr_type

```
typedef enum
{
    IP6_ADDR_TYPE_MANUAL = 0,
    IP6_ADDR_TYPE_AUTOCONFIGURABLE = 1
} rtcs6_if_addr_type;
```

IP6_ADDR_TYPE_MANUAL

手动设置 IPv6 地址。

IP6_ADDR_TYPE_AUTOCONFIGURABLE

通过自动配置设置 IPv6 地址。

9.2.52 RTCSMIB_VALUE

该结构说明了用于读取 MIB 节点值及其类型的函数。

```
typedef struct rtcsmib_value {
    uint32_t TYPE;
```

```
    void *PARAM;
} RTCSMIB_VALUE;
```

TYPE

值的类型。为非叶节点时忽略。可为以下之一：

- RTCSMIB_NODETYPE_INT_CONST：常数整数。
- RTCSMIB_NODETYPE_INT_PTR：指向整数的指针。
- RTCSMIB_NODETYPE_INT_FN：指向函数返回整数的指针。
- RTCSMIB_NODETYPE_UINT_CONST：常数无符号整数。
- RTCSMIB_NODETYPE_UINT_PTR：指向无符号整数的指针。
- RTCSMIB_NODETYPE_UINT_FN：指向函数返回无符号整数的指针。
- RTCSMIB_NODETYPE_DISPSTR_PTR：显示字符串（可打印字符串）。
- RTCSMIB_NODETYPE_DISPSTR_FN：指向函数返回字符串的指针。
- RTCSMIB_NODETYPE_OCTSTR_FN：八位组字符串（八位组序列）。
- RTCSMIB_NODETYPE_OID_PTR：指向 MIB 节点的指针，其中 OID 用作节点值。
- RTCSMIB_NODETYPE_OID_FN：指向返回节点的函数的指针。然后将该节点的 OID 用作节点值。

PARAM

指针/值/函数取决于类型。

9.2.53 SMTP_EMAIL_ENVELOPE 结构

该结构存储成功传递电子邮件所需的信息。在 RFC 中为 SMTP 地址。在 *rtcs_smtp.h* 文件中进行说明。

```
typedef struct smtp_email_envelope
{
    char    *from;
    char    *to;
}SMTP_EMAIL_ENVELOPE, * SMTP_EMAIL_ENVELOPE_PTR;
```

from

包含作为参数传递到 MAIL FROM 命令的字符串。

to

包含作为参数传递到 RCPT TO 命令的字符串。

9.2.54 SMTP_PARAM_STRUCT 结构

```
typedef struct smtp_param_struct
{
```

```
    SMTP_EMAIL_ENVELOPE envelope;
    char *text;
    struct sockaddr* server;
    char *login;
    char *pass;
    bool auth_req;
}SMTP_PARAM_STRUCT, * SMTP_PARAM_STRUCT_PTR;
```

envelope

在第 SMTP_EMAIL_ENVELOPE 章的结构中说明的 SMTP 地址。

ext

要发送的电子邮件正文。其中必须包含完全格式化的电子邮件消息。以下为消息的最小内容和格式:

```
"From: <>\r\n"
"To: <>\r\n"
"Subject: \r\n"
"Date: \r\n\r\n"
```

关于消息格式和用法的详细示例，请参见文件*shell\source\rtcs\sh_smtp.c*。

server

用于发送电子邮件的 SMTP 服务器。将在 SMTP 端口上创建套接字并连接用于与该服务器进行通信。

login

用于 SMTP 验证的用户名。可为 NULL，则使用无验证。

pass

用于 SMTP 验证的密码。如果为 NULL，则使用验证时将向服务器发送空密码。

9.2.55 sockaddr_in

用于套接字端点标识符的结构。

```
typedef struct sockaddr_in
{
  uint16_t  sin_family;
  uint16_t  sin_port;
  in_addr  sin_addr;
} sockaddr_in;
```

sin_family

地址系列类型。

sin_port

端口号。

sin_addr

IP 地址。

9.2.56 sockaddr_in6

用于 IPv6 套接字端点标识符的结构。

```
typedef struct sockaddr_in6
{
    uint16_t    sin6_family;
    uint16_t    sin6_port;
    in6_addr          sin6_addr;
    uint32_t    sin6_scope_id;
}sockaddr_in6;
```

sin6_family

地址系列类型。设为 AF_INET6

sin6_port

传输层端口号（按主机字节顺序）。

sin6_addr

128 位 IPv6 地址。

sin6_scope_id

范围区索引（接口标识符）。

9.2.57 sockaddr

用于由 IPv4 和 IPv6 支持的套接字端点标识符的结构。

```
#if RTCSCFG_ENABLE_IP6
    typedef struct sockaddr
    {
        uint16_t         sa_family;
        char          sa_data[22];
    } sockaddr;
#else
    #if RTCSCFG_ENABLE_IP4
        #define sockaddr   sockaddr_in
        #define sa_family   sin_family
    #endif
#endif
```

sa_family

表示地址格式的代码。这用于标识之后跟随的数据格式。

sa_data

实际套接字地址数据取决于格式。长度也由格式确定。

每个地址格式均具有以“**AF_**”开头的符号名。

AF_INET

这确定了与互联网命名空间匹配的地址格式。

AF_INET6

这与 AF_INET 类似，也可参见 IPv6 协议。

AF_UNSPEC

这决定了不存在特殊地址格式。

9.2.58 TCP_STATS

指向该结构的指针由 TCP_stats() 返回。

```
typedef struct {
  uint32_t              ST_RX_TOTAL;
  uint32_t              ST_RX_MISSED;
  uint32_t              ST_RX_DISCARDED;
  uint32_t              ST_RX_ERRORS;
  uint32_t              ST_TX_TOTAL;
  uint32_t              ST_TX_MISSED;
  uint32_t              ST_TX_DISCARDED;
  uint32_t              ST_TX_ERRORS;
  RTCS_ERROR_STRUCT    ERR_RX;
  RTCS_ERROR_STRUCT    ERR_TX;
  uint32_t              ST_RX_BAD_PORT;
  uint32_t              ST_RX_BAD_CHECKSUM;
  uint32_t              ST_RX_BAD_OPTION;
  uint32_t              ST_RX_BAD_SOURCE;
  uint32_t              ST_RX_SMALL_HDR;
  uint32_t              ST_RX_SMALL_DGRAM;
  uint32_t              ST_RX_SMALL_PKT;
  uint32_t              ST_RX_BAD_ACK;
  uint32_t              ST_RX_BAD_DATA;
  uint32_t              ST_RX_LATE_DATA;
  uint32_t              ST_RX_OPT_MSS;
  uint32_t              ST_RX_OPT_OTHER;
  uint32_t              ST_RX_DATA;
  uint32_t              ST_RX_DATA_DUP;
  uint32_t              ST_RX_ACK;
  uint32_t              ST_RX_ACK_DUP;
  uint32_t              ST_RX_RESET;
  uint32_t              ST_RX_PROBE;
  uint32_t              ST_RX_WINDOW;

  uint32_t              ST_RX_SYN_EXPECTED;
  uint32_t              ST_RX_ACK_EXPECTED;
  uint32_t              ST_RX_SYN_NOT_EXPECTED
  uint32_t              ST_RX_MULTICASTS;
  uint32_t              ST_TX_DATA;
  uint32_t              ST_TX_DATA_DUP;
  uint32_t              ST_TX_ACK;
  uint32_t              ST_TX_ACK_DELAYED;
  uint32_t              ST_TX_RESET;
  uint32_t              ST_TX_PROBE;
```

```
  uint32_t              ST_TX_WINDOW;
  uint32_t              ST_CONN_ACTIVE;
  uint32_t              ST_CONN_PASSIVE;
  uint32_t              ST_CONN_OPEN;
  uint32_t              ST_CONN_CLOSED;
  uint32_t              ST_CONN_RESET;
  uint32_t              ST_CONN_FAILED;
    uint32_t              ST_CONN_ABORTS;
} TCP_STATS, * TCP_STATS_PTR;
```

ST_RX_TOTAL

已接收的数据包总数。

ST_RX_MISSED

由于缺少资源而丢弃的输入数据包。

ST_RX_DISCARDED

由于所有其他原因而丢弃的输入数据包。

ST_RX_ERRORS

在处理输入数据包时检测到内部错误。

ST_TX_TOTAL

已发送的数据包总数。

ST_TX_MISSED

由于缺少资源而丢弃的发送数据包。

ST_TX_DISCARDED

由于所有其他原因而丢弃的发送数据包。

ST_TX_ERRORS

在尝试发送数据包时检测到内部错误。

ERR_RX

接收错误信息。

ERR_TX

发送错误信息。

以下内容包含在 *ST_RX_DISCARDED* 中。

ST_RX_BAD_PORT

具有目标端口零的段码。

ST_RX_BAD_CHECKSUM

具有无效校验和的段码。

ST_RX_BAD_OPTION

具有无效选项的段码。

ST_RX_BAD_SOURCE

具有无效源的段码。

ST_RX_SMALL_HDR

头文件太小的段码。

ST_RX_SMALL_DGRAM

段码小于头文件。

ST_RX_SMALL_PKT

段码大于帧。

ST_RX_BAD_ACK

接收到未发送数据的 ACK。

ST_RX_BAD_DATA

接收到窗口外的数据。

ST_RX_LATE_DATA

在关闭后接收到数据。

ST_RX_OPT_MSS

MSS 选项置位的段码。

ST_RX_OPT_OTHER

具有其他选项的段码。

ST_RX_DATA

已接收数据段码。

ST_RX_DATA_DUP

已重复接收数据。

ST_RX_ACK

已接收 ACK。

ST_RX_ACK_DUP

已重复接收 ACK。

ST_RX_RESET

已接收 RST 段码。

ST_RX_PROBE

已接收窗口探测器。

ST_RX_WINDOW

已接收窗口更新。

ST_RX_SYN_EXPECTED

未接收期望的 SYN。

ST_RX_ACK_EXPECTED

未接收期望的 ACK。

ST_RX_SYN_NOT_EXPECTED

已接收不期望的 SYN。

ST_RX_MULTICASTS

组播数据包。

ST_TX_DATA

已发送数据段码。

ST_TX_DATA_DUP

已重新发送数据段码。

ST_TX_ACK

已发送仅 ACK 段码。

ST_TX_ACK_DELAYED

已发送延迟 ACK。

ST_TX_RESET

已发送 RST 段码。

ST_TX_PROBE

已发送窗口探测器。

ST_TX_WINDOW

已发送窗口更新。

ST_CONN_ACTIVE

主动打开操作。

ST_CONN_PASSIVE

被动打开操作。

ST_CONN_OPEN

已建立连接。

ST_CONN_CLOSED

正常关闭操作。

ST_CONN_RESET

非正常关闭操作。

ST_CONN_FAILED

打开操作失败。

ST_CONN_ABORTS

中止操作。

9.2.59 TELNETCLN_CALLBACK

用于 Telnet 客户端回调的原型。

```
typedef void (TELNETCLN_CALLBACK)(void *param);
```

9.2.60 TELNETCLN_CALLBACKS_STRUCT

结构包含 Telnet 的所有客户端回调。

```
typedef struct telnetcln_callbacks_struct
{
```

```
    TELNETCLN_CALLBACK *on_connected;
    TELNETCLN_CALLBACK *on_disconnected;
    void               *param;
}TELNETCLN_CALLBACKS_STRUCT;
```

on_connected

当客户端连接时调用回调。如果该回调设为 NULL，则忽略。

on_disconnected

当客户端从服务器断开连接时调用回调。如果该回调设为 NULL，则忽略。

param

用于 Telnet 客户端回调的参数。当设为 NULL 时忽略。

9.2.61 TELNETCLN_PARAM_STRUCT

```
typedef struct telnetcln_param_struct
{
    sockaddr                   sa_remote_host;
    bool                       use_nagle;
    MQX_FILE_PTR               fd_in;
    MQX_FILE_PTR               fd_out;
    TELNETCLN_CALLBACKS_STRUCT callbacks;
}TELNETCLN_PARAM_STRUCT;
```

sa_remote_host

Sockaddr 结构填充了关于客户端尝试连接的远程 Telnet 主机的信息。

use_nagle

标志用于确定是否将 Nagle 算法用于连接服务器。

fd_in

文件指针作为 Telnet 客户端输入使用。

fd_out

文件指针作为 Telnet 客户端输出使用。

callbacks

结构包含回调和回调参数。

9.2.62 TELNETSRV_PARAM_STRUCT

该结构作为 FTPSRV_init()函数的参数使用。

```
typedef struct telnetsrv_param_struct
{
    uint16_t              af;
    unsigned short        port;
    #if RTCSCFG_ENABLE_IP4
    in_addr               ipv4_address;
    #endif
    #if RTCSCFG_ENABLE_IP6
    in6_addr              ipv6_address;
    uint32_t              ipv6_scope_id;
    #endif
    uint32_t              max_ses;        /* maximal sessions count */
    bool                  use_nagle;      /* enable/disable nagle algorithm for server
sockets */
    uint32_t              server_prio;    /* server main task priority */
    TELNET_SHELL_FUNCTION shell;          /* Pointer to shell to run */
    void                  *shell_commands;/* Pointer to shell commands */
} TELNETSRV_PARAM_STRUCT;
```

af

服务器使用的地址系列。可能值为：AF_INET（使用 IPv4）、AF_INET6（使用 IPv6）、

AF_INET | AF_INET6（使用 IPv4 和 IPv6）。默认情况下，服务器将使用所有使能的地址系列。

port

用于监听的端口。默认值由 RFC 定义为 23。

ipv4_address

用于监听的 IPv4 地址。只有当使能 IPv4 时，才使用该变量。服务器默认将监听所有可用地址。

ipv6_address

用于监听的 IPv6 地址。只有当使能 IPv6 时，才使用该变量。默认值为 in6addr_any。

ipv6_scope_id

IPv6 的范围 ID（接口标识）。默认值为 0。

max_ses

同时连接到服务器的最大用户数。默认值由

宏 TELNETSRVCFG_DEF_SES_CNT 定义(2)。

use_nagle

设为 TRUE 以使能用于服务器套接字的 NAGLE 算法。默认为 FALSE - 禁止 NAGLE。

server_prio

服务器任务的优先级。由服务器创建的所有任务（服务器任务和会话任务）均运行于该优先级。

默认值由宏 RTCSCFG_TELNETSRV_DEF_SERVER_PRIO 定义。

shell

在客户端连接后要运行的函数。

shell_commands

指向用户用来连接 Telnet 服务器的可用命令表的指针。

9.2.63 TFTPCLN_PARAM_STRUCT

该结构作为 TFTPCLN_connect()函数的参数使用。

```
typedef struct tftpcln_param_struct
{
    sockaddr                  sa_remote_host;
    TELNETCLN_DATA_CALLBACK   recv_callback;
    TELNETCLN_DATA_CALLBACK   send_callback;
    TELNETCLN_ERROR_CALLBACK error_callback;
}TFTPCLN_PARAM_STRUCT;
```

sa_remote_host

关于远程主机客户端连接的信息。这通常由 getaddrinfo()函数创建。该变量为强制变量。

recv_callback

当接收到数据包时调用回调。该变量可以为 NULL。

send_callback

当发送数据包时调用回调。该变量可以为 NULL。

error_callback

当在传输数据过程中发生错误时调用回调。该变量可以为 NULL。

9.2.64 TELNETCLN_DATA_CALLBACK

TFTP 客户端发送/接收回调的原型。

```
typedef void(*TELNETCLN_DATA_CALLBACK)(uint32_t data_length);
```

data_length

发送/接收数据包的长度。

9.2.65 TELNETCLN_ERROR_CALLBACK

TFTP 客户端错误回调的原型。

```
typedef void(*TELNETCLN_ERROR_CALLBACK)(uint16_t error_code, char* error_string);
```

error_code

服务器报告的错误数值。

error_string

由服务器发送的用于说明错误的字符串。

9.2.66 TFTPSRV_PARAM_STRUCT

该结构作为 FTPSRV_init()函数的参数使用。

```
typedef struct tftpsrv_param_struct
{
uint16_t af;
uint16_t port;
#if RTCSCFG_ENABLE_IP4
in_addr ipv4_address;
#endif
#if RTCSCFG_ENABLE_IP6
in6_addr ipv6_address;
uint32_t ipv6_scope_id;
#endif
uint32_t max_ses;
uint32_t server_prio;
char* root_dir;
}TFTPSRV_PARAM_STRUCT;
```

af

服务器使用的地址系列。可能值为：AF_INET（使用 IPv4)、AF_INET6（使用 IPv6)、AF_INET | AF_INET6（使用 IPv4 和 IPv6)。

port

用于监听的端口。默认值由 RFC 定义为 21。

ipv4_address

用于监听的 IPv4 地址。只有当使能 IPv4 时，才使用该变量。默认值为 0（监听所有地址)。

ipv6_address

用于监听的 IPv6 地址。只有当使能 IPv6 时，才使用该变量。默认值为 in6addr_any。

ipv6_scope_id

IPv6 的范围 ID（接口标识）。默认值为 0。

max_ses

同时连接到服务器的最大用户数。默认值由宏 RTCSCFG_TFTPSRV_SES_CNT 定义

server_prio

服务器任务的优先级。由服务器创建的所有任务（服务器任务和会话任务）均运行于该优先级。默认值由宏 RTCSCFG_TFTPSRV_SERVER_PRIO 定义。

root_dir

服务器根目录。TFTP 客户端只能访问该目录及其子目录下的文件。

9.2.67 UDP_STATS

指向该结构的指针由 UDP_stats() 返回。

```
typedef struct {
  uint32_t             ST_RX_TOTAL;
  uint32_t             ST_RX_MISSED;
  uint32_t             ST_RX_DISCARDED;
  uint32_t             ST_RX_ERRORS;
  uint32_t             ST_TX_TOTAL;
  uint32_t             ST_TX_MISSED;
  uint32_t             ST_TX_DISCARDED;
  uint32_t             ST_TX_ERRORS;
  RTCS_ERROR_STRUCT   ERR_RX;
  RTCS_ERROR_STRUCT   ERR_TX;
  uint32_t             ST_RX_BAD_PORT;
  uint32_t             ST_RX_BAD_CHECKSUM;
  uint32_t             ST_RX_SMALL_DGRAM;
  uint32_t             ST_RX_SMALL_PKT;
  uint32_t             ST_RX_NO_PORT;
} UDP_STATS, * UDP_STATS_PTR;
```

ST_RX_TOTAL

已接收的数据包总数。

ST_RX_MISSED

由于缺少资源而丢弃的输入数据包。

ST_RX_DISCARDED

由于所有其他原因而丢弃的输入数据包。

ST_RX_ERRORS

在处理输入数据包时检测到内部错误。

ST_TX_TOTAL

已发送的数据包总数。

ST_TX_MISSED

由于缺少资源而丢弃的发送数据包。

ST_TX_DISCARDED

由于所有其他原因而丢弃的发送数据包。

ST_TX_ERRORS

在尝试发送数据包时检测到内部错误。

ERR_RX

接收错误信息。

ERR_TX

发送错误信息。

以下状态包含在 *ST_RX_DISCARDED* 中。

ST_RX_BAD_PORT

目标端口零的数据报。

ST_RX_BAD_CHECKSUM

无效校验和的数据报。

ST_RX_SMALL_DGRAM

数据报小于头文件。

ST_RX_SMALL_PKT

数据报大于帧。

ST_RX_NO_PORT

用于已关闭端口的数据报。

9.2.68 WS_DATA_STRUCT

WebSocket 数据结构。

```
typedef struct ws_data_struct
{
    uint8_t      *data_ptr;
    uint32_t     length;
    WS_DATA_TYPE type;
}WS_DATA_STRUCT;
```

data_ptr

指向发送/接收数据的指针。

length

待发送/接收数据的长度。

type

数据类型（WS_DATA_INVALID、WS_DATA_TEXT 和 WS_DATA_BINARY）。

9.2.69 WS_PLUGIN_STRUCT

结构定义了用于 WebSocket 插件的回调和参数。

```
typedef struct ws_plugin_struct
{
    WS_CALLBACK_FN on_connect;
    WS_CALLBACK_FN on_message;
    WS_CALLBACK_FN on_error;
    WS_CALLBACK_FN on_disconnect;
    void*          cookie;
}WS_PLUGIN_STRUCT;
```

on_connect

指针指向当客户端连接服务器时调用的函数。

on_message

指针指向当从客户端接收消息时调用的函数。

on_error

指针指向当发生错误时调用的函数。

on_disconnect

指针指向当客户端与服务器断开连接时调用的函数。

cookie

回调参数。

9.2.70 WS_USER_CONTEXT_STRUCT

结构作为参数传递给所有 WebSocket 回调。

```
typedef struct ws_user_context_struct
{
    uint32_t            handle;
    WS_ERROR_CODE       error;
    WS_DATA_STRUCT      data;
    uint32_t            fin_flag;
}WS_USER_CONTEXT_STRUCT;
```

handle

WebSocket 句柄。

error

错误代码，如果发生错误。

data

描述数据的结构。

fin_flag

指示消息结束的标志。

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

本文档中的信息仅供系统和软件实施方使用 Freescale 产品。 本文并未明示或者暗示授予利用本文档信息进行设计或者加工集成电路的版权许可。 Freescale 保留对此处任何产品进行更改的权利，恕不另行通知。

Freescale 对其产品在任何特定用途方面的适用性不做任何担保、表示或保证，也不承担因为应用程序或者使用产品或电路所产生的任何责任，明确拒绝承担包括但不局限于后果性的或附带性的损害在内的所有责任。 Freescale 的数据表和/或规格中所提供的“典型”参数在不同应用中可能并且确实不同，实际性能会随时间而有所变化。 所有运行参数，包括“经典值”在内，必须经由客户的技术专家对每个客户的应用程序进行验证。 Freescale 未转让与其专利权及其他权利相关的许可。 Freescale 销售产品时遵循以下网址中包含的标准销售条款和条件：freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, AltiVec, C-5, CodeTest, CodeWarrior, ColdFire, ColdFire+, C-Ware, Energy Efficient Solutions logo, Kinetis, mobileGT, PowerQUICC, Processor Expert, QorIQ, Qorivva, StarCore, Symphony, and VortiQa are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Airfast, BeeKit, BeeStack, CoreNet, Flexis, Layerscape, MagniV, MXC, Platform in a Package, QorIQ Qonverge, QUICC Engine, Ready Play, SafeAssure, SafeAssure logo, SMARTMOS, Tower, TurboLink, Vybrid, and Xtrinsic are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© 2015 Freescale Semiconductor, Inc.

© 2015 飞思卡尔半导体有限公司

Document Number MQXRTCSUG
Revision 2, 04/2015