



PRODUCT MANUAL

Dynamic C[®]

TCP/IP User's Manual

Volume 1

019-0143h

The latest revision of this manual is available on the Rabbit Web site,
www.rabbit.com, for free, unregistered download.

Dynamic C TCP/IP User's Manual

Volume 1

Part Number 019-0143 • Printed in the U.S.A

Digi International Inc. © 2007-2009 • All rights reserved.

Digi International Inc. reserves the right to make changes and improvements to its products without providing notice.

Trademarks

Rabbit and Dynamic C[®] are registered trademarks of Digi International Inc.

Windows[®] is a registered trademark of Microsoft Corporation

Table of Contents

1	Introduction.....	1			
2	TCP/IP Initialization	3			
	2.1 TCP/IP Stack Configuration.....	3			
	2.1.1 Multiple Interface Support	3			
	2.1.2 Interface Selection Macros	5			
	2.1.2.1 Link Layer Drivers	6			
	2.1.3 Single Interface Support	7			
	2.1.3.1 Configuration Macros for Link Layer Driver - Single Interface	7			
	2.1.4 TCP/IP Stack Initialization	7			
	2.2 Interface Configuration	8			
	2.2.1 Configuration Overview	8			
	2.2.2 Sources of Configuration Information 9				
	2.2.2.1 Predefined Configurations	9			
	2.2.2.2 Static Configuration	10			
	2.2.2.3 Dynamic Configuration via the Network	11			
	2.2.2.4 Runtime Configuration Using if- config()	12			
	2.2.2.5 Directed Ping	13			
	2.2.2.6 Remote Configuration via Ad- vanced Device Discovery Protocol (AD- DP)	14			
	2.2.2.7 Console Configuration Via Zconsole.lib	14			
	2.2.3 Media Access Control (MAC) Address 14				
	2.3 Dynamically Starting and Stopping Interfaces. 15				
	2.3.1 Testing Interface Status	15			
	2.3.2 Bringing an Interface Up	15			
	2.3.3 Bringing an Interface Down	16			
	2.4 Setting Up Wi-Fi Interfaces.....	16			
	2.4.1 Wi-Fi Compile Time Configuration	17			
	2.4.1.1 Infrastructure, Open (No En- cryption) Configuration	17			
	2.4.1.2 Ad-hoc, Open (No Encryption) Configuration	18			
	2.4.1.3 Infrastructure, WEP Encryption Configuration	19			
	2.4.1.4 Infrastructure, WPA/TKIP En- cryption Configuration, Pre-Shared Key 20				
	2.4.1.5 Infrastructure, WPA2/CCMP Encryption Configuration, Pre-Shared Key	21			
	2.4.1.6 Infrastructure, WPA Enterprise using EAP-TLS and CCMP Encryption 22				
	2.4.1.7 Infrastructure, WPA Enterprise using PEAP and TKIP Encryption ..	24			
	2.4.1.8 Specifying a Pre-Shared Key	25			
	2.4.1.9 Ad-hoc, WPA/TKIP or WPA2/CCMP Encryption Configura- tion	25			
	2.4.2 Wi-Fi Runtime Configuration	25			
	2.4.2.1 Runtime Configuration Starting with Dynamic C 10.40	25			
	2.5 Setting Up PPP Interfaces	25			
	2.5.1 PPP over Asynchronous Serial	25			
	2.5.2 PPP over Ethernet	26			
	2.6 Configuration Macro Reference.....	26			
	2.6.1 Removing Unnecessary Functions	26			
	2.6.2 Including Additional Functions	27			
	2.6.3 BOOTP/DHCP Control Macros	27			
	2.6.4 Buffer and Resource Sizing	28			
	2.6.5 Network Configuration Prior to Dynamic C 7.30	31			
	2.6.6 Network Configuration Starting with Dynamic C 7.30	31			
	2.6.7 Time-Outs and Retry Counters	33			
	2.6.8 Program Debugging	34			
	2.6.9 Miscellaneous Macros	35			
	2.6.9.1 TOS and TTL	35			
	2.6.10 Wi-Fi Configuration Macros	36			
3	TCP and UDP Socket Interface	41			
	3.1 What is a Socket?	42			
	3.1.1 Port Numbers	42			
	3.2 Allocating TCP and UDP Sockets	43			
	3.2.1 Allocating Socket Buffers	43			
	3.2.2 Socket Buffer Sizes	44			
	3.2.2.1 User-Supplied Buffers	44			
	3.3 Opening TCP Sockets	44			
	3.3.1 Passive Open	44			
	3.3.2 Active Open	45			
	3.3.3 Waiting for Connection Establishment 45				
	3.3.4 Specifying a Listen Queue	46			
	3.4 TCP Socket Functions.....	46			
	3.4.1 Control Functions for TCP Sockets	46			
	3.4.2 Status Functions for TCP Sockets .	47			
	3.4.3 I/O Functions for TCP Sockets	48			
	3.5 UDP Socket Overview	49			
	3.6 UDP Socket Functions (7.05 and later).....	50			
	3.6.1 Control Functions for UDP Sockets .	50			
	3.6.2 Status Function for UDP Sockets ..	50			

3.6.3 I/O Functions for UDP Sockets	50	6.2 IGMP	82
3.7 UDP Socket Functions (pre 7.05)	51	6.3 Multicast Macros	82
3.7.1 I/O Functions for UDP Sockets	51	7 PPP Driver	83
3.7.2 Opening and Closing a UDP Socket	51	7.1 PPP Libraries	83
3.7.3 Writing to a UDP Socket	51	7.2 External Modem Library	84
3.7.4 Reading From a UDP Socket	52	7.3 Operation Details for PPP over Serial	84
3.7.5 Porting Programs from the older UDP		7.3.1 The Modem Interface	84
API to the new UDP API	52	7.3.1.1 Rabbit Pin Connections to Mo-	
3.8 Skeleton Program	53	dem	85
3.8.1 TCP/IP Stack Initialization	53	7.3.2 Flow Control	85
3.8.2 Packet Processing	54	7.4 Operation Details for PPPoE	86
3.9 TCP/IP Daemon: tcp_tick()	54	7.5 Link Control Protocol Options	86
3.9.1 tcp_tick() for Robust Applications	54	7.6 Configuring PPP	87
3.9.2 Global Timer Variables	55	7.6.1 Serial Port Selection	87
3.10 State-Based Program Design	55	7.6.2 PPPoE Port Selection	87
3.10.1 Blocking vs. Non-Blocking	56	7.6.3 ifconfig() Options for PPP	88
3.10.1.1 Non-Blocking Functions	56	7.6.4 ifconfig() Options for Serial PPP	89
3.10.1.2 Blocking Functions	56	7.6.4.1 Additional Rules for Send/Ex-	
3.11 TCP and UDP Data Handlers	57	pect Scripts	90
3.11.1 UDP Data Handler	59	7.6.5 Starting and Stopping PPP Interfaces	91
3.11.2 TCP Data Handler	59	8 Function Reference	93
3.12 Multitasking and TCP/IP	61	_abort_socks	95
3.12.1 μ C/OS-II	61	arpcache_create	96
3.12.2 Cooperative Multitasking	62	arpcache_flush	97
4 Optimizing TCP/IP Performance	65	arpcache_hwa	98
4.1 DBP and Sizing of TCP Buffers	66	arpcache_iface	99
4.2 TCP Performance Tuning	68	arpcache_ipaddr	100
4.2.1 The Nagle Algorithm	68	arpcache_load	101
4.2.2 Time-Out Settings	69	arpcache_search	103
4.2.2.1 Time-Out Setting Constants	70	arp_getArpData	104
4.2.3 Reserved Ports	72	arp_getArpGateData	104
4.2.4 Type of Service (TOS)	73	_arp_resolve	105
4.2.5 ARP Cache Considerations	73	arpresolve_check	106
4.3 Writing a Fast UDP Request/Response Server	74	arpresolve_ipaddr	107
4.4 Tips and Tricks for TCP Applications	74	arpresolve_start	108
4.4.1 Bulk Loader Applications	75	arpresolve_start_iface	109
4.4.2 Casual Server Applications	76	_arp_send_gratuitous	110
4.4.3 Master Controller Applications	76	aton	111
4.4.4 Web Server Applications	76	_chk_ping	112
4.4.5 Protocol Translator Applications	76	dhcp_acquire	113
5 Network Addressing: ARP & DNS	77	dhcp_get_timezone	114
5.1 ARP Functions	77	dhcp_release	115
5.2 Configuration Macros for ARP	77	getdomainname	116
5.3 DNS Functions	79	gethostid	117
5.4 Configuration Macros for DNS Lookups	79	gethostname	118
6 IGMP and Multicasting	81	getpeername	119
6.1 Multicasting	81	getsockname	120
6.1.1 Multicast Addresses	81		
6.1.2 Host Group Membership	81		

htonl	121	router_for	182
htons	122	router_for_iface	183
ifconfig	123	router_print	185
ifdown	146	router_printall	186
ifpending	147	_send_ping	187
ifstatus	148	setdomainname	188
ifup	149	sethostid	189
inet_addr	151	sethostname	190
inet_ntoa	152	sock_abort	191
ip_iface	153	sock_alive	192
ip_print_ifs	154	sock_aread	193
ip_timer_expired	155	sock_awrite	194
ip_timer_init	156	sock_axread	195
is_valid_iface	157	sock_axwrite	196
ModemClose	158	sock_bytesready	197
ModemConnected	158	sock_close	198
ModemExpect	159	sock_dataready	199
ModemHangup	159	sockerr	200
ModemInit	160	sock_error	201
ModemOpen	160	sock_established	202
ModemReady	161	sock_fastread	203
ModemRinging	161	sock_fastwrite	204
ModemSend	162	sock_flush	205
ModemStartPPP	162	sock_flushnext	206
multicast_joingroup	163	sock_getc	207
multicast_leavegroup	164	sock_gets	208
ntohl	165	sock_iface	209
ntohs	166	sock_init	210
pd_getaddress	167	sock_init_or_exit	211
pd_havelink	168	sock_mode	212
pd_powerdown	169	sock_noflush	214
pd_powerup	170	sock_perror	215
_ping	171	sock_pread	216
PPPactive	172	sock_putc	217
PPPnegotiateIP	172	sock_puts	218
PPPsetAuthenticate	173	sock_rbleft	219
PPPsetAuthenticator	173	sock_rbsize	220
PPPshutdown	174	sock_rused	221
psocket	174	sock_read	222
resolve	175	sock_readable	223
resolve_cancel	176	sock_recv	224
resolve_name_check	177	sock_recv_from	226
resolve_name_start	178	sock_recv_init	227
rip	179	sock_resolved	228
router_add	180	sock_set_tos	229
router_del_all	181	sock_set_ttl	230
router_delete	181	sockstate	231

sock_tbleft	232
sock_tbsize	233
sock_tbusd	234
sock_tick	235
sock_wait_closed	236
sock_wait_established.....	237
sock_waiting	238
sock_wait_input	239
sock_writable	240
sock_write	241
sock_xfastread.....	242
sock_xfastwrite	243
sock_yield	244
tcp_clearreserve	245
tcp_config.....	246
tcp_extlisten	247
tcp_extopen	248
tcp_keepalive	249
tcp_listen	250
tcp_open	252
tcp_reserveport.....	254
tcp_tick.....	255
udp_bypass_arp.....	256
udp_close	257
udp_extopen	258
udp_open.....	260
udp_peek	262
udp_recv	263
udp_recvfrom.....	264
udp_send	265
udp_sendto	266
udp_waitopen	267
udp_waitsend	268
udp_xsendto	269
virtual_eth	270
wifi_ssid_to_str	271
Index	273

1. Introduction

This manual is intended for embedded system designers and support professionals who are using a Rabbit-based controller board. Most of the information contained here is meant for use with Ethernet- or WiFi-enabled boards, but using only serial communication is also an option. Knowledge of networks and TCP/IP (Transmission Control Protocol/Internet Protocol) is assumed. For an overview of these two topics a separate manual is provided, *An Introduction to TCP/IP*. A basic understanding of HTML (HyperText Markup Language) is also assumed. For information on this subject, there are numerous sources on the Web and in any major book store.

The Dynamic C implementation of TCP/IP comprises several libraries. The main library is `DCRTCP.LIB`. As of Dynamic C 7.05, this library is a light wrapper around `DNS.LIB`, `IP.LIB`, `NET.LIB`, `TCP.LIB` and `UDP.LIB`. These libraries implement DNS (Domain Name Server), IP, TCP, and UDP (User Datagram Protocol). This, along with the libraries `ARP.LIB`, `ICMP.LIB`, `IGMP.LIB` and `PPP.LIB` are the transport and network layers of the TCP/IP protocol stack.

The Dynamic C libraries that implement application-layer protocols are:

- `BOOTP.LIB`
- `FTP_SERVER.LIB`
- `FTP_CLIENT.LIB`
- `HTTP.LIB`
- `HTTP_CLIENT.LIB`
- `POP3.LIB`
- `SMNP.LIB`
- `SMTP.LIB`
- `TFTP.LIB`
- `VSERIAL.LIB`

Except for `BOOTP` (which is described in this manual) the application-layer protocols are described in *Dynamic C TCP/IP, User's Manual, Vol. 2*.

All user-callable functions are listed and described in their appropriate chapter. Example programs throughout the manual illustrate the use of all the different protocols. The sample code also provides templates for creating servers and clients of various types.

To address embedded system design needs, additional functionality has been included in Dynamic C's implementation of TCP/IP. There are step-by-step instructions on how to create HTML forms, allowing remote access and manipulation of information. There is also a serial-based console that can be used with TCP/IP to open up legacy systems for additional control and monitoring. The console may also be used for configuration when a serial port is available. The console and HTML forms are discussed in the *Dynamic C TCP/IP User's Manual, Vol. 2*.

Multiple interfaces are supported starting with Dynamic C version 7.30.

2. TCP/IP Initialization

This chapter describes the configuration macros, data structures and functions used to configure and initialize the Dynamic C TCP/IP stack. Starting with Dynamic C version 7.30, the stack supports multiple interfaces. Interface configuration is described in “Interface Configuration” on page 8.

The Dynamic C TCP/IP stack supports IP version 4. Although multiple interfaces are supported, the TCP/IP stack does not support packet routing at the IP level.

2.1 TCP/IP Stack Configuration

You need to know certain things to configure the stack. You need to know which interfaces will be used and how many. You also need to determine the necessary software functionality. For example, will there be DNS lookups? Are TCP and UDP protocols both necessary? Will DHCP be used? The ability to remove unneeded features via conditional compilation has been enhanced starting with Dynamic C 7.30. This is accomplished with the configuration macros described in “Removing Unnecessary Functions” on page 26 and “Including Additional Functions” on page 27.

2.1.1 Multiple Interface Support

The supported interfaces are:

- Ethernet
- PPP (Point-to-Point Protocol) over a serial link
- PPP over Ethernet
- Wi-Fi (802.11b, 802.11g)

The interfaces must be on distinct, non-overlapping subnets. In particular, each interface must be assigned a unique IP address, known as the “home IP address” for that interface.

The interfaces available to your application will depend on the hardware configuration of the target board. All Rabbit-based boards have at least four asynchronous serial ports, so PPP over serial is always available. Many boards have an Ethernet port. If an Ethernet port is available, then it may be used for normal Ethernet or PPP over Ethernet (PPPoE). No Rabbit-based board has more than one Ethernet port, however Dynamic C 7.30 contains support for a second Ethernet interface if and when such a board becomes available.

Your application uses configuration macros to select the interface(s) to use for TCP/IP. Each hardware interface will have an interface number assigned. The interface number is not used directly; instead, your application should use the macros defined for this purpose. If you are writing general-purpose routines, then you should include `#ifdef` tests for the interface macro if you need to refer to it. This is because the macros are not necessarily defined for non-existent interfaces. The macros are:

IF_ETH0, IF_ETH1

These macros represent Ethernet ports that are not using PPP. `IF_ETH0` refers to the first Ethernet port and `IF_ETH1` to the second.

IF_PPPOE0, IF_PPPOE1

These macros represent Ethernet ports used for PPP over Ethernet. `IF_PPPOE0` refers to the first (and currently only) Ethernet port.

PPPoE and regular Ethernet can co-exist on the same Ethernet hardware. PPPoE effectively sets up a virtual point-to-point link between two devices on the same Ethernet LAN segment.

IF_PPP0, IF_PPP1, IF_PPP2, IF_PPP3, IF_PPP4, IF_PPP5

These macros represent asynchronous serial ports used for PPP. `IF_PPP0` always refers to serial port A, `IF_PPP1` refers to serial port B, etc. Most boards will avoid using serial port A, since it is most often used for Dynamic C debugging and program download.

IF_PPPX

This is an alias for the “first” PPP interface. The first PPP interface is selected as the first valid interface in the following order: `IF_PPPOE0`, `IF_PPPOE1`, `IF_PPP0`, `IF_PPP1`, etc. through to `IF_PPP5`.

IF_WIFI0, IF_WIFI1

These macros represent Wi-Fi interfaces. Only `IF_WIFI0` is supported at this time.

IF_DEFAULT

This is an alias for the “default” interface. You can explicitly define this macro prior to including `dcrtcp.lib` to select a default interface. The Dynamic C TCP/IP libraries do not make use of `IF_DEFAULT` with the important exception of DHCP. DHCP only works on the default interface.

If you do not explicitly define `IF_DEFAULT`, it is chosen as the first valid interface in the following order: `IF_PPPX` (see above), `IF_WIFI0`, `IF_ETH0`.

If you explicitly define `IF_DEFAULT`, then you must define it to a hard-coded integer value, not one of the `IF_*` macros, since the `IF_*` macros are not defined until `dcrtcp.lib` is included. Since the actual numbers assigned to each interface depend on the values of the `USE_*` macros, you must be careful when doing this. The only time you may want to explicitly define `IF_DEFAULT` is when you are using both PPP and non-PPPoE Ethernet, and you want to use DHCP on the Ethernet interface.

IF_ANY

This is not an interface as such. It is a special value used to denote “any” or “all” interfaces, where applicable. This macro should be used only where a function documents that its use is acceptable. For example, the `tcp_extlisten()` function accepts `IF_ANY` as an interface parameter, which tells it to listen for incoming connections on any available interface.

2.1.2 Interface Selection Macros

As each physical interface has its own macro, each type of interface has a corresponding macro. The macro value determines which physical interfaces of the same type will be supported by the stack. Setting the macro to zero disables support for that type of interface, i.e., no physical interfaces of that type will be supported. If the macros are not defined in the application program, they will be set to zero internally.

USE_ETHERNET

This macro allows support of non-PPPoE Ethernet. It can be set to 0x01, 0x02 or 0x03. Most boards only support 0x01, meaning the first non-PPPoE Ethernet device. Boards with two Ethernet devices can set this macro to 0x02, referring to the second Ethernet device, or 0x03 to allow use of both devices.

USE_PPP_SERIAL

This macro allows support of PPP over asynchronous serial. It can be set to:

- 0x01 (serial port A)
- 0x02 (serial port B)
- 0x04 (serial port C)
- 0x08 (serial port D)
- 0x10 (serial port E, available on Rabbit 3000 and above)
- 0x20 (serial port F, available on Rabbit 3000 and above)

Or any bitwise combination of these values.

Serial port C is the default, but you may use any of the others. Please note that if you use serial port A (the programming port) Dynamic C will not be able to communicate with the target. You may also need to define other macros to allow correct functioning of the serial port hardware, e.g., hardware flow control.

USE_PPPOE

This macro allows support of PPP over Ethernet. It is set in the same way as `USE_ETHERNET`. The bitmask indicates which Ethernet devices are to be used for PPP over Ethernet.

USE_WIFI

This macro allows support of Wi-Fi. It can be set to 0x01 or 0x02.

2.1.2.1 Link Layer Drivers

The `USE_*` configuration macros described in “Interface Selection Macros” on page 5 cause the appropriate link layer drivers to be included. If none of the `USE_*` macros are defined and the macro `PKTDRV` is also not defined, `realtek.lib` will be used. Some board types cause a driver other than `realtek.lib` to be used, e.g., if the board is the RCM3200 or the RCM3210, the packet driver library `asix.lib` will replace `realtek.lib`.

The following table tells which link layer drivers will be used when a `USE_*` macro is defined to a value greater than zero.

Table 2.1 Libraries Included When `USE_*` Macro Value > Zero

Configuration Macro	Realtek.lib*	Ppp.lib	Ppplink.lib	Pppoe.lib	WiFiG.lib
<code>USE_ETHERNET</code>	yes	no	no	no	no
<code>USE_PPP_SERIAL</code>	no	yes	yes	no	no
<code>USE_PPPOE</code>	yes	yes	no	yes	no
<code>USE_WIFI</code>	no	no	no	no	yes

* or a substitute packet driver library based on board type

As the table reveals, using PPP over Ethernet causes `realtek.lib`, `ppp.lib` and `pppoe.lib` to be included. Multiple drivers may also be included by defining multiple interfaces.

The following macros are defined for applications to perform conditional compilation that depends on the drivers actually included:

- `USING_ETHERNET`
- `USING_PPP_SERIAL`
- `USING_PPPOE`
- `USING_WIFI`
- `USING_WIFIG`

These macros are always defined, but will have a zero value if the driver was not included. Thus, the conditional compilation should use the `#if` operator, not `#ifdef`. For example,

```
#if USING_PPP_SERIAL
    // Do something special for PPP over serial
#endif
```

The value assigned to the `USING_*` macro is the number of hardware interfaces of that type that are available. On a Rabbit 2000 board, `USING_PPP_SERIAL` will be defined to 4 or 0. On a Rabbit 3000 board, the value will be 6 or 0.

An additional macro, `USING_PPP`, is also defined if any of the PPP-type interfaces are in use. Unlike the above macros, this macro is either defined or not defined, so the correct test is `#ifdef`.

2.1.3 Single Interface Support

Backwards compatibility exists for applications compiled with earlier versions of Dynamic C. If none of the `USE_*` macros are defined, then the old behavior (pre-Dynamic C 7.30) is used, which is to include one, and only one, link layer driver.

2.1.3.1 Configuration Macros for Link Layer Driver - Single Interface

Do not define either of these macros if any of the `USE_*` macros are defined.

PKTDRV

This macro specifies the packet driver to use. Include one of the following statements in your application.

```
#define PKTDRV "realtek.lib"    // To use Ethernet
#define PKTDRV "ppp.lib"      // To use PPP (serial or Ethernet)
```

PPPOE

This macro is defined to use PPP over Ethernet when `PKTDRV` is set to `ppp.lib`. For other packet drivers, this define has no effect (but should not be defined in order to avoid problems with future Dynamic C releases).

```
#define PPPOE
```

2.1.4 TCP/IP Stack Initialization

The function `sock_init()` must be called near the start of your `main()` function in order to initialize the TCP/IP stack. The return value from `sock_init()` must indicate success before calling any other TCP/IP functions, with the possible exception of `ifconfig()`.

IMPORTANT: If you are using μ C/OS-II, then you must ensure that `OSInit()` is called before calling `sock_init()`.

The function `sock_init()` performs the following actions, and does not return until complete (or an error was encountered):

- Calls subsystem initialization for ARP, TCP, UDP and DNS (if applicable).
- Tests to see whether `sock_init()` was run previously. If so, then it returns OK. Otherwise, the following steps are executed.
- Initialize the packet driver; basically this resets the hardware and clears out the packet receive buffer pool.
- Clears the router and other server tables.
- Interfaces are initialized using the settings specified in the `IFCONFIG_*` macros or predefined configurations.

WiFi takes longer to initialize than the other supported interfaces, and potentially much longer if there is no AP within range (which is analogous to not plugging in the Ethernet cable). If all of the above completed successfully, the return code is set to 0. Otherwise, the return code will be non-zero, however you can still proceed if the return code is 2 since this indicates that DHCP failed but fallbacks were used. Other return codes indicate that the network is not usable.

After `sock_init()` returns OK, the non-PPPoE Ethernet interface should be ready for traffic if it is intended to be up initially. PPP interfaces may not be fully started even if requested to be up initially. PPP interfaces can take a substantial amount of time to come up, especially if modem dial-out is in use. You can wait for a particular interface to come up by polling the interface status using `ifstatus()` or, preferably, `ifpending()`.

2.2 Interface Configuration

Prior to Dynamic C version 7.30, only a single network interface was supported. Configuration of the interface was performed by defining a set of macros, such as `MY_IP_ADDRESS`¹, as well as by calling various configuration functions such as `sethostid()`.

With Dynamic C 7.30's support of multiple interfaces, the macro-style configuration becomes impractical, and the configuration functions generally would require an additional parameter, the interface number. Version 7.30 implements a slightly different method of configuration, but maintains compatibility with the old style of configuration for simple applications that require only a single interface.

It is recommended that new applications use the new style of configuration, even if multiple interface support is not required. This will ease the integration of future Dynamic C upgrades.

2.2.1 Configuration Overview

To run the TCP/IP stack, a host (i.e., the controller board) needs to know its unique home IP address for each interface. Interfaces that connect to broadcast networks (e.g., Ethernet or Wi-Fi) must also have a netmask assigned. The combination of IP address and netmask describes the so-called subnet which is addressable on that interface. The subnet basically describes the community of host addresses that can talk directly to this host, without requiring data to pass through a packet router. Point-to-point links only need an IP address, since there is only one other host by definition.

IP address and netmask are the most important configuration items; however, many other items are needed for successful networking. For anything but strictly local communication, a router or gateway host must be known. The router has the important task of forwarding messages between the local host and the outside world (i.e., hosts that are not on the local subnet). Routers are associated with particular interfaces. Each interface will generally require a different router; however, in the majority of cases only one interface will actually be used to talk to non-local hosts so only one router will be required to service all requests for non-local host addresses.

Some of the configuration items are not specific to any particular interface. For example, DNS (Domain Name System) servers are known by their IP address. DNS servers are used to translate human-readable domain names (e.g., `www.rabbit.com`) into machine-readable IP addresses.

1. The configuration macros `MY_IP_ADDRESS` and `MY_NETMASK` have been deprecated in favor of `_PRIMARY_STATIC_IP` and `_PRIMARY_NETMASK`, respectively.

2.2.2 Sources of Configuration Information

The Dynamic C TCP/IP stack obtains configuration information from one or more of the following sources:

- Use one of the predefined configurations in `tcp_config.lib`; static or dynamic.
- Macro definitions before `#use "dcrtcp.lib"`; static configuration.
- Bootstrap network protocols such as BOOTP and DHCP; dynamic configuration.
- Runtime function calls such as `ifconfig()` (version 7.30) and `sethostid()` (previous versions).
- “Directed ping” IP address assignment (new in version 7.30).
- Console-based configuration, e.g., `zconsole.lib`.

As application designer, you have to decide which of these configuration techniques is right for your project. Entirely static configuration is typically used for initial application development and testing. Most of the TCP/IP sample programs use static configuration for simplicity in getting started. Applications which are intended for real-world use should allow at least one form of dynamic configuration. The particular form of configuration that is supported will depend on the complexity of the application, as well as the expected network or operational environment in which the application will run.

2.2.2.1 Predefined Configurations

Since networking configuration can be fairly complicated, Dynamic C version 7.30 has the concept of “canned” or predefined configurations. This has the advantage of reducing the number of macro definitions at the top of each TCP/IP program, as well as eliminating the need for copy/paste of a lot of settings from one program to the next.

Using the predefined configurations is very easy: simply `#define` a single macro (called `TCPCONFIG`) at the top of each program. The macro is defined to an integer, which selects one of the predefined configurations in `tcp_config.lib`. For example:

```
#define TCPCONFIG 1
#use "dcrtcp.lib"
```

causes the first predefined configuration to be used.

Most of the sample TCP/IP programs refer to one of the predefined configurations. It is fairly likely (unfortunately) that none of the configurations will work with your network. For example, the default IP address of “10.10.6.100” may not be allowed on your LAN. If this is the case, you can define the default IP address in your application or in the “Defines” tab of the Project Options dialog box.

To configure within your program, copy and paste the following into your code and modify as necessary:

```
#define TCPCONFIG          1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK  "255.255.255.0"
#define MY_GATEWAY        "10.10.6.1"
#define MY_NAMESERVER     "10.10.6.1"
```

To configure within the Project Options dialog box, copy and paste the following into the Defines window and modify as necessary:

```
TCPCONFIG          = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK   = "255.255.255.0";
MY_GATEWAY         = "10.10.6.1";
MY_NAMESERVER      = "10.10.6.1";
```

Another way to change the default configuration is to create a library called `custom_config.lib`. In this library, you can place your own custom configurations which will not be overwritten by Dynamic C (since this is not a released library).

To create `custom_config.lib`, you can use `tcp_config.lib` as a template. Modify the definitions to suit your network environment. You must change the configuration numbers (i.e., the value of `TCPCONFIG`) to be greater than or equal to 100. Numbers less than 100 are expected to be in `tcp_config.lib`; numbers over 99 cause `custom_config.lib` to be included.

If you are using a Dynamic C version prior to 9.30, then the other thing you must do before using your own custom configurations is to add the library name (`custom_config.lib`) to the `lib.dir` file in the base Dynamic C installation directory. This is just a text file, which you can edit with the Dynamic C text editor. Locate the line that contains “`tcp_config.lib`.” Repeat this line, and modify one of the line copies to point to your `custom_config.lib` file. You will not have to restart Dynamic C for this change to take effect. A new release of Dynamic C will overwrite the `lib.dir` file, so you will need to perform this edit for each release.

Starting with Dynamic C 9.30, `lib.dir` references entire directories automatically, so it is not necessary to edit it for the addition of the custom library.

To use custom configurations that you define, the only thing necessary in each sample program is to change the definition of the `TCPCONFIG` macro to indicate the appropriate configuration e.g.,

```
#define TCPCONFIG 100
#use "dcrtcp.lib"
```

2.2.2.2 Static Configuration

This is conceptually the easiest means of configuration; however it is primarily suitable for testing purposes (or possibly as a fallback in case other configuration techniques do not yield a result in a reasonable amount of time).

Prior to version 7.30, the (only) interface was configured by defining a fixed set of macros before including `dcrtcp.lib`. The most common definitions were limited to: `MY_IP_ADDRESS`, `MY_NETMASK`, `MY_GATEWAY` and `MY_NAMESERVER`.

At runtime, the functions, `tcp_config()`, `sethostid()` and `sethostname()` override the configuration macros.

Version 7.30 still allows use of these macros for backwards compatibility; however, it is recommended that the new style of static configuration be used for new applications. The new configuration style uses macros called `IFCONFIG_*`, where “*” is replaced by the interface name e.g., `IFCONFIG_ETH0` for the

first Ethernet port. `IFCONFIG_ALL` contains configuration items that are not specific to any particular interface.

The value of the `IFCONFIG_*` macro is actually a list of items in the syntactic form of a C parameter list. For example, if the old style configuration (for Ethernet) was

```
#define MY_IP_ADDRESS "10.10.6.100"  
#define MY_NETMASK "255.255.255.0"  
#define MY_GATEWAY "10.10.6.1"
```

then the new replacement would be

```
#define IFCONFIG_ETH0 \  
    IFS_IPADDR, aton("10.10.6.100"), \  
    IFS_NETMASK, aton("255.255.255.0"), \  
    IFS_ROUTER_SET, aton("10.10.6.1"), \  
    IFS_UP
```

The replacement looks more complex, but this is because the macro value must be valid C syntax for a parameter list. The `IFS_UP` parameter at the end of the above example is a new feature for interfaces: they can be dynamically brought up and down. The default state for an interface is “down,” which is why an explicit `IFS_UP` is required. The backslashes at the end of each line are used to continue the macro definition over more than one line.

The format of the static initialization macros will make more sense if you examine the documentation for the `ifconfig()` function. You will see that the macro definition is merely “plugged in” to the parameter list for an `ifconfig()` call.

2.2.2.3 Dynamic Configuration via the Network

The Dynamic C TCP/IP stack supports DHCP (Dynamic Host Configuration Protocol) or BOOTP (Bootstrap Protocol) for dynamic configuration. DHCP is a more modern replacement for BOOTP, which was originally designed to support bootstrap of diskless workstations. Use of these protocols can completely eliminate the need for static configuration.

The library `BOOTP.LIB` allows a target board to be a BOOTP or DHCP client. The protocol used depends on what type of server is installed on the local network. BOOTP and DHCP servers are usually centrally located on a local network and operated by the network administrator. Note that initialization may take longer when using DHCP as opposed to static configuration, but this depends on your server. Also note that if the interface goes down for any reason, the application will need to bring the interface back up and redo any changes to the default configuration, such as setting `IFS_DHCP_TIMEOUT`. This behavior differs from the behavior of a PC, which will automatically bring the interface back up.

Both protocols allow a number of configuration parameters to be sent to the client, including:

- client’s IP address
- net mask
- list of gateways
- host and default domain names
- list of name servers

BOOTP assigns permanent IP addresses. DHCP can “lease” an IP address to a host, i.e., assign the IP address for a limited amount of time. There are two user-callable functions regarding IP address leases

`dhcp_release()` and `dhcp_acquire()` (described in “Function Reference” on page 93). In addition, there are a number of macros and global variables available for modifying behavior and obtaining information. Please see “BOOTP/DHCP Control Macros” on page 27 for details.

As of Dynamic C 7.30, DHCP or BOOTP can be used only on the default interface i.e., the interface that is specified by the value of `IF_DEFAULT`. If you are using more than one interface then you should ensure that `IF_DEFAULT` is set correctly.

To successfully use DHCP configuration, ensure all of the following conditions are met. (Only the first condition applies prior to 7.30.)

- `#define USE_DHCP` before including `dcrtcp.lib`.
- Ensure `IF_DEFAULT` is indicating the desired interface.
- Define an `IFCONFIG_*` macro to include the `IFS_DHCP` parameter ID.

For example, if the Ethernet interface is to be used for DHCP, the following code is required for DHCP:

```
#define USE_DHCP
#define IF_DEFAULT 0           // not necessary unless also using PPP
#define IFCONFIG_ETH0 IFS_DHCP, 1, IFS_UP
#include "dcrtcp.lib"
```

You may also use the predefined configuration number 3, which is DHCP:

```
#define TCPCONFIG 3
#include "dcrtcp.lib"
```

This configuration sets all required macros for DHCP (or BOOTP) to work. Naturally, there must be a DHCP server available on the interface. The DHCP server must be set up to contain all the required configuration options, however setting up the DHCP server is outside the scope of this document, since there are many different DHCP servers in use.

The sample program `Samples\tcpip\dhcp.c` uses dynamic configuration in a basic TCP/IP program that will initialize the TCP/IP interface, and allow the device to be “pinged” from another computer on the network. It demonstrates DHCP features, such as releasing and re-acquiring IP addresses and downloading a configuration file.

2.2.2.4 Runtime Configuration Using `ifconfig()`

`ifconfig()` is a function introduced in version 7.30. This function does many things, and is the recommended replacement for some of the functions marked as “deprecated” (including `tcp_config()`). `ifconfig()` performs most of the work for all the other configuration techniques. For example, static configuration (via the `IFCONFIG_*` macros) basically calls `ifconfig()` with the specified parameters substituted in.

`ifconfig()` takes a variable number of parameters, like `printf()`, however the parameter list is terminated with the special `IFS_END` symbol. For example, to use `ifconfig()` to set the same parameters as described above for the static configuration:

```
ifconfig(IF_ETH0, IFS_IPADDR, aton("10.10.6.100"),
        IFS_NETMASK, aton("255.255.255.0"),
        IFS_ROUTER_SET, aton("10.10.6.1"),
        IFS_UP,
        IFS_END);
```

Note that this is the same as substitution of the `IFCONFIG_*` macro e.g.,

```
ifconfig(IF_ETH0, IFCONFIG_ETH0, IFS_END);
```

`ifconfig()` is also used to obtain current configuration items at runtime. For example,

```
longword ipaddr;
ifconfig(IF_ETH0, IFG_IPADDR, &ipaddr, IFS_END);
```

gets the current IP address of the first Ethernet interface into the variable `ipaddr`.

The first parameter of `ifconfig()` is the interface number. For certain settings, this can also be `IF_ANY`, which means apply the settings to all applicable interfaces. The parameters following the first are an arbitrary number of tuples consisting of a parameter identifier followed by the value(s) for that parameter (if any). The list of parameters must be terminated by a special identifier, `IFS_END`. See the documentation for `ifconfig()` for a complete list of parameter identifiers with their expected values.

2.2.2.5 Directed Ping

This style of configuration, also known as ICMP configuration, is limited to setting the IP address of the interface. It only works on non-PPPoE Ethernet interfaces. To specify directed ping configuration, use the `IFS_ICMP_CONFIG` parameter ID in a call to `ifconfig()` or in the definition of the `IFCONFIG_*` macro for the interface. For example

```
#define IFCONFIG_ETH0 IFS_ICMP_PING, 1
```

for a static configuration, or

```
ifconfig(IF_ETH0, IFS_ICMP_CONFIG, 1, IFS_END);
```

at runtime. Note that you can use both directed ping and DHCP on the same interface, but directed ping is not limited to just the default interface. If both directed ping and DHCP are allowed on a particular interface, the first one “wins.”

Directed ping works as follows. The interface is brought up, but has no assigned IP address so it cannot be used for normal traffic. If the interface receives an ICMP echo request (i.e., ping) which is directed to the interface’s MAC address, then the destination IP address in the ICMP packet is assigned to the interface as its home IP address. After that point, the interface is configured and is available for normal traffic.

The weakness of directed ping is that only the IP address is provided. The netmask must be pre-configured or obtained by other means. Technically, directed ping violates some tenets of the Internet standards, however, it can be useful in controlled environments.

The MAC address of the board must be known (see below) for directed ping to work. The host which initiates the ICMP echo request must have its ARP table statically configured with the target MAC address. On Unix and Windows hosts, the appropriate command sequence is

```
arp -s <IP address> <MAC address>
```

followed by

```
ping <IP address>
```

The actual format of the MAC address depends on the operating system. Most hosts will recognize a format like “00-09-A0-20-00-99”. The IP address is in dotted decimal notation.

Once the interface is configured by directed ping (or DHCP), then further directed ping or DHCP configurations for that interface are not allowed. If desired, at runtime you can issue

```
ifconfig(IF_ETH0, IFS_ICMP_CONFIG_RESET, IFS_END);
```

to allow another directed ping configure.

2.2.2.6 Remote Configuration via Advanced Device Discovery Protocol (ADDP)

The `ADDP.lib` library contains a UDP-based daemon that will listen for requests from an ADDP client such as the Digi Device Discovery utility included in the Utilities\ADDP directory of your Dynamic C installation. The Discovery utility runs on a Windows PC and can discover ADDP-enabled devices on the local network and modify their network configuration (with the appropriate password).

The sample program `Samples\tcpip\addp.c` demonstrates using the Discovery utility to remotely configure a device that can store its network configuration in the UserBlock (non-volatile storage), and then use that stored configuration at startup to configure the network interface.

2.2.2.7 Console Configuration Via Zconsole.lib

The `zconsole.lib` library contains routines for allowing an external (serial or telnet) terminal to issue configuration commands. Basically, the commands call `ifconfig()` to perform the actual requests or obtain information.

Using a “dumb terminal” connection over a serial port presents no special difficulties for network configuration. Using telnet over the internet obviously requires a working TCP stack to begin with. This is still useful in the case that one of the other configuration techniques can at least get to a working state. For example, directed ping can assign an IP address. You could then use the same host to telnet into the new IP address in order to set other items like the netmask and router.

2.2.3 Media Access Control (MAC) Address

Rarely, ISPs require that the user provide them with a MAC address for their device. Run the utility program, `Samples\tcpip\display_mac.c`, to display the MAC address of your controller board.

The MAC address is also required for directed PING configure, as well as some other bootstrap techniques. MAC addresses are often written as a sequence of six two-digit hexadecimal numbers, separated by colons e.g., 00:90:20:33:00:A3. This distinguishes them from IP addresses, which are written with dotted decimal numbers.

MAC addresses are completely unrelated to IP addresses. IP addresses uniquely identify each host on the global Internet. MAC addresses uniquely identify Ethernet hardware on a particular Ethernet LAN

segment. Although only technically required to be unique on a LAN segment, in practice MAC addresses are globally unique and can thus be used to uniquely identify a particular Ethernet adapter.

The usual reason for an ISP requiring a MAC address is if the ISP uses DHCP to dynamically assign IP addresses. Most ISPs use PPP (Point to Point Protocol) which does not care about MAC addresses. DHCP can use the MAC address to determine that the same device is connecting, and assign it the same IP address as before.

2.3 Dynamically Starting and Stopping Interfaces

Starting with version 7.30, Dynamic C allows interfaces to be individually brought up and down by calling the `ifup()`, `ifdown()` or `ifconfig()` functions. The initial desired state of the interface is specified using the `IFCONFIG_*` macros. By default, interfaces are not brought up when `sock_init()` is called at boot time. Only if the `IFCONFIG_*` macro contains an `IFS_UP` directive will the interface be brought up at boot time.

Most applications should not need to dynamically change the interface status. The exception to this may be PPP over serial interfaces, where a modem is used to dial out to an ISP on demand.

2.3.1 Testing Interface Status

There are two functions for testing the current status of an interface: `ifstatus()` and `ifpending()`. The function `ifstatus()` merely returns a boolean value indicating whether the interface is up. If the return value is true (non-zero), then the interface is ready for normal TCP/IP communications. Otherwise, the interface is not yet available; it may either be down, or in the process of coming up.

`ifpending()` gives more information: its return value indicates not only the current state, but also if the state is in the process of changing. This function is very handy when using WiFi; in particular, it can be used to make sure it is okay to call the `wifi_ioctl()` function for most commands.

If your application needs to check the interface status, which is recommended for PPP over serial or PPPoE, then it can either poll the status using the above functions, or it can register a callback function which is automatically called whenever the interface changes status.

To register a callback function, you call `ifconfig()` with the `IFS_IF_CALLBACK` as the parameter identifier, and the address of your callback function as the parameter value.

2.3.2 Bringing an Interface Up

You can call `ifup()`, or `ifconfig()` with the `IFS_UP` parameter identifier. The advantage of using `ifconfig()` is that you can specify an interface number of `IF_ANY`, which brings all interfaces up together.

When the `ifup()` call returns, the interface may not have completed coming up. This is notably the case for PPP interfaces, which require a number of protocol negotiation packets to be sent and received. In addition, PPP over serial may require additional time to reset a modem, dial out to an ISP, and possibly respond to the ISP's login procedure. All this could take considerable time, so the `ifup()` function does not wait around for the process to complete, to allow the application to proceed with other work.

On return from the `ifup()` call, an application must test for completion using the functions described in the previous section.

For the interface to come up completely, your application must call `tcp_tick()` regularly while waiting for it. If you can afford to block until the interface is up, then use code similar to the following:

```
ifup(IF_PPP2);
// Wait for the interface to have any status other than "down coming up."
while (ifpending(IF_PPP2) == 1) tcp_tick();
if (ifstatus(IF_PPP2))
    printf("PPP2 is up now.\n");
else
    printf("PPP2 failed to come up.\n");
```

2.3.3 Bringing an Interface Down

You can call `ifdown()`, or `ifconfig()` with the `IFS_DOWN` parameter identifier. The advantage of using `ifconfig()` is that you can specify an interface number of `IF_ANY`, which brings all interfaces down together.

As for `ifup()`, `ifdown()` does not necessarily complete immediately on return. PPP requires link tear-down messages to be sent to the peer and acknowledged. Thus, similar considerations apply to bringing an interface down as they do for bringing it up.

`ifdown()` will always succeed eventually. Unlike `ifup()`, which can possibly fail to bring the interface up, `ifdown()` will always eventually return success i.e., it is not possible for an interface to be left "hanging up." If the PPP link tear-down does not get an acknowledgment from the peer, then the process times out and the link is forced down.

2.4 Setting Up Wi-Fi Interfaces

Wi-Fi has some specific configuration macros in addition to the basic TCP/IP macros that define the IP address, netmask, gateway and nameserver. This section will describe the setup of Wi-Fi, both at compile time and runtime. The configuration macros used in the code in this section are described in "Wi-Fi Configuration Macros" on page 36.

The Wi-Fi interface is considered "up" (`ifpending()` returns that the interface is up and the link LED is on) in the following conditions:

- If connecting to an AP using no encryption, then the link is "up" when 802.11 association has completed.
- If connecting to an AP with TKIP/WPA or CCMP/WPA2 encryption, the link is "up" after the 802.11i handshake has completed successfully. This means that the pre-shared key must be correct before the link is considered up.
- If connecting to an AP with WEP encryption, then the link is "up" when 802.11 association has completed, whether or not the WEP key is correct. This means that a WEP connection can show as being "up" even though the key is wrong. This is because, unlike TKIP or CCMP, WEP does not provide a way of knowing whether or not the key used is correct.

2.4.1 Wi-Fi Compile Time Configuration

The examples given here cover both Wi-Fi modes: ad-hoc and infrastructure. Note that a Wi-Fi interface can be configured for DHCP support but the examples in this section will show only static configuration.

2.4.1.1 Infrastructure, Open (No Encryption) Configuration

The simplest way to configure a Wi-Fi network interface is to use an open configuration. This means that no encryption or authentication is used. This example is configured for infrastructure mode, which means that you will also need a Wi-Fi access point or Wi-Fi router. Note that your wireless access point will need to match these settings (in particular, the SSID must be the same, and the AP must not have encryption enabled).

To configure within your program, copy and paste the following into your sample program and modify as necessary:

```
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define IFC_WIFI_SSID "rabbitTest"
#define IFC_WIFI_ROAM_ENABLE 1
#define IFC_WIFI_ROAM_BEACON_MISS 20
#define IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCT
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_NONE
```

Alternatively, you can configure within the Defines window. Here is an example to copy and paste into the Defines window and modify as necessary:

```
TCPCONFIG = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK = "255.255.255.0";
MY_GATEWAY = "10.10.6.1";
MY_NAMESERVER = "10.10.6.1";
IFC_WIFI_SSID = "rabbitTest";
IFC_WIFI_ROAM_ENABLE = 1;
IFC_WIFI_ROAM_BEACON_MISS = 20;
IFC_WIFI_MODE = IFPARAM_WIFI_INFRASTRUCTURE;
IFC_WIFI_REGION = IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION = IFPARAM_WIFI_ENCR_NONE;
```

2.4.1.2 Ad-hoc, Open (No Encryption) Configuration

Another simple way to configure a Wi-Fi network interface is to use an open configuration in ad-hoc mode. An ad-hoc Wi-Fi network does not require an access point or Wi-Fi router to operate. Instead, the Wi-Fi devices communicate directly with each other. This is useful for an isolated Wi-Fi network, but it is not typically used when devices need to communicate outside of their own network. Note that all Wi-Fi devices on an ad-hoc network need to match these settings (in particular, the SSID and the channel number must be the same).

To configure within your program, copy and paste the following into your sample program and modify as necessary:

```
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define _WIFI_SSID "rabbitTest"
#define IFC_WIFI_ROAM_ENABLE 1
#define IFC_WIFI_ROAM_BEACON_MISS 20
#define IFC_WIFI_CHANNEL 1
#define IFC_WIFI_MODE IFPARAM_WIFI_ADHOC
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_NONE
```

Alternatively, you can configure within the Defines window. Here is an example to copy and paste into the Defines window and modify as necessary:

```
TCPCONFIG = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK = "255.255.255.0";
MY_GATEWAY = "10.10.6.1";
MY_NAMESERVER = "10.10.6.1";
IFC_WIFI_SSID = "rabbitTest";
IFC_WIFI_ROAM_ENABLE = 1;
IFC_WIFI_ROAM_BEACON_MISS = 20;
IFC_WIFI_CHANNEL = 1;
IFC_WIFI_MODE = IFPARAM_WIFI_ADHOC;
IFC_WIFI_REGION = IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION = IFPARAM_WIFI_ENCR_NONE;
```


2.4.1.3 Infrastructure, WEP Encryption Configuration

WEP is an encryption method for Wi-Fi networks. It is now considered insecure, but it is commonly supported by Wi-Fi devices. To use WEP, all devices including the access point or Wi-Fi router will need to be configured with the same WEP keys (as well as the same SSID). WEP allows multiple keys to be defined; all devices will need to be configured to use the same key number.

WEP keys are used for WEP shared key authentication and for WEP encryption. The Wi-Fi driver stores up to 4 different WEP keys (0 to 3) of either 5 or 13 bytes.

To configure within your program, copy and paste the following into your sample program and modify as necessary:

```
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define IFC_WIFI_SSID "rabbitTest"
#define IFC_WIFI_ROAM_ENABLE 1
#define IFC_WIFI_ROAM_BEACON_MISS 20
#define IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCTURE
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_WEP
#define IFC_WIFI_WEP_KEYNUM 0
#define IFC_WIFI_WEP_KEY0_HEXSTR "0123456789abcdef0123456789"

// alternate method to define a WEP key
#define IFC_WIFI_WEP_KEY0_BIN \
0x01, 0x23, 0x45, 0x67, 0x89, 0xab,0xcd, 0xef,\
0x01, 0x23, 0x45, 0x67, 0x89
```

Alternatively, you can configure within the Defines window. Here is an example to copy and paste into the Defines window and modify as necessary:

```
TCPCONFIG = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK = "255.255.255.0";
MY_GATEWAY = "10.10.6.1";
MY_NAMESERVER = "10.10.6.1";
IFC_WIFI_SSID = "rabbitTest";
IFC_WIFI_ROAM_ENABLE = 1;
IFC_WIFI_ROAM_BEACON_MISS = 20;
IFC_WIFI_MODE = IFPARAM_WIFI_INFRASTRUCTURE;
IFC_WIFI_REGION = IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION = IFPARAM_WIFI_ENCR_WEP;
IFC_WIFI_WEP_KEYNUM = 0;
IFC_WIFI_WEP_KEY0_HEXSTR = "0123456789abcdef0123456789";
```

2.4.1.4 Infrastructure, WPA/TKIP Encryption Configuration, Pre-Shared Key

Wi-Fi Protected Access (WPA) along with Temporal Key Integrity Protocol (TKIP) replaces the less secure WEP encryption method. The Rabbit Wi-Fi driver supports WPA encryption¹ in pre-shared key (PSK) mode. This key can be specified as a 256-bit key or as a passphrase that is expanded into a key. The code for WPA/TKIP is compiled when you define the macro `WIFI_USE_WPA`.

To configure within your program, copy and paste the following into your sample program and modify as necessary:

```
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define WIFI_USE_WPA
#define IFC_WIFI_SSID "rabbitTest"
#define IFC_WIFI_ROAM_ENABLE 1
#define IFC_WIFI_ROAM_BEACON_MISS 20
#define IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCTURE
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_TKIP
```

You will also need to define WPA's pre-shared key mode. This can be done by specifying a passphrase that the driver will expand to a 256-bit key or by specifying the 256-bit key itself. Note that expanding a passphrase takes a significant amount of time at startup, up to about 20 seconds. Define `WIFI_VERBOSE_PASSPHRASE` to see status messages when the key is generated from the passphrase.

To specify the WPA passphrase, you will need something like the following:

```
#define IFC_WIFI_WPA_PSK_PASSPHRASE "myPassphrase"
```

To specify the WPA Pre-Shared Key itself, use the following:

```
#define IFC_WIFI_WPA_PSK_HEXSTR \
"1122334455667788990011223344556677889900112233445566778899001122"
```

Alternatively, you can configure within the Defines window:

```
TCPCONFIG = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK = "255.255.255.0";
MY_GATEWAY = "10.10.6.1";
MY_NAMESERVER = "10.10.6.1";
WIFI_USE_WPA;
IFC_WIFI_SSID = "rabbitTest";
IFC_WIFI_ROAM_ENABLE = 1;
IFC_WIFI_ROAM_BEACON_MISS = 20;
IFC_WIFI_MODE = IFPARAM_WIFI_INFRASTRUCTURE;
IFC_WIFI_REGION = IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION = IFPARAM_WIFI_ENCR_TKIP;
```

1. In general usage the term "WPA encryption" sometimes refers to WPA using TKIP.

To specify a WPA passphrase or Pre-Shared Key:

```
IFC_WIFI_WPA_PSK_PASSPHRASE = "myPassphrase" ;
```

To specify the WPA Pre-Shared Key itself:

```
IFC_WIFI_WPA_PSK_HEXSTR =  
"1122334455667788990011223344556677889900112233445566778899001122" ;
```

2.4.1.5 Infrastructure, WPA2/CCMP Encryption Configuration, Pre-Shared Key

WPA2 is a more secure replacement for WEP. This implementation uses the Advanced Encryption Standard (AES) based algorithm, also known as CCMP (Counter Mode with Cipher Block Chaining Message Authentication Code Protocol) cipher suite. The Rabbit Wi-Fi driver supports CCMP/WPA2 encryption with Pre-Shared Keys (PSK). These keys can be specified as a 256-bit key or as a passphrase that is expanded into a key. Note that the code for this encryption method is not compiled unless you define the macros `WIFI_USE_WPA` and `WIFI_AES_ENABLED`.

To configure within your program, copy and paste the following into your sample program and modify as necessary for your situation:

```
#define TCPCONFIG 1  
#define _PRIMARY_STATIC_IP "10.10.6.100"  
#define _PRIMARY_NETMASK "255.255.255.0"  
#define MY_GATEWAY "10.10.6.1"  
#define MY_NAMESERVER "10.10.6.1"  
#define WIFI_USE_WPA  
#define WIFI_AES_ENABLED  
#define IFC_WIFI_SSID "rabbitTest"  
#define IFC_WIFI_ROAM_ENABLE 1  
#define IFC_WIFI_ROAM_BEACON_MISS 20  
#define IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCTURE  
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS  
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_CCMP
```

You will also need to define WPA2's pre-shared key mode. This can be done by specifying a passphrase that the driver will expand to a 256-bit key or by specifying the 256-bit key itself. Note that expanding a passphrase takes a significant amount of time at startup, up to about 20 seconds. Define `WIFI_VERBOSE_PASSPHRASE` to see status messages when the key is generated from the passphrase.

To specify the WPA2 passphrase, you will need something like the following:

```
#define IFC_WIFI_WPA_PSK_PASSPHRASE "myPassphrase"
```

To specify the WPA2 Pre-Shared Key itself, use the following:

```
#define IFC_WIFI_WPA_PSK_HEXSTR \  
"1122334455667788990011223344556677889900112233445566778899001122"
```

Alternatively, you can configure within the Defines window:

```
TCPCONFIG 1;
_PRIMARY_STATIC_IP "10.10.6.100";
_PRIMARY_NETMASK "255.255.255.0";
MY_GATEWAY "10.10.6.1";
MY_NAMESERVER "10.10.6.1";
WIFI_USE_WPA;
WIFI_AES_ENABLED;
IFC_WIFI_SSID "rabbitTest";
IFC_WIFI_ROAM_ENABLE 1;
IFC_WIFI_ROAM_BEACON_MISS 20;
IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCTURE;
IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_CCMP;
```

Again, you will need to specify a WPA2 passphrase or Pre-Shared Key. To specify the WPA2 passphrase, use the following:

```
IFC_WIFI_WPA_PSK_PASSPHRASE = "myPassphrase";
```

To specify the WPA Pre-Shared Key itself, use the following:

```
IFC_WIFI_WPA_PSK_HEXSTR = \
"1122334455667788990011223344556677889900112233445566778899001122";
```

2.4.1.6 Infrastructure, WPA Enterprise using EAP-TLS and CCMP Encryption

To configure within your program, copy and paste the following into your sample program and modify as necessary:

```
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define WIFI_USE_WPA
#define WIFI_AES_ENABLED
#define IFC_WIFI_SSID "rabbitTest"
#define IFC_WIFI_ROAM_ENABLE 1
#define IFC_WIFI_ROAM_BEACON_MISS 20
#define IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCTURE
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_CCMP
// The above defines are identical to using CCMP with PSK
```

```

#define WPA_USE_EAP WPA_USE_EAP_TLS
#define IFC_WIFI_WPA_PROTOCOL IFPARAM_WIFI_WPA_PROTOCOL_WPA2
#define ximport "certs/root.pem" root_ca_cert
#define ximport "certs/my_client.pem" my_client_cert
#define IFC_WIFI_CLIENT_CERT_XIM my_client_cert, my_client_cert
#define IFC_WIFI_CA_CERT_XIM root_ca_cert
#define IFC_WIFI_IDENTITY "MyClientIdent"
#define SSL_USE_AES

```

Because it is necessary to ximport SSL certificates for use with this encryption method, it is not possible to completely configure WPA Enterprise with EAP/TLS and CCMP using the Project Defines. However, as long as you include something like these two #ximport statements in your program:

```

#define ximport "certs/root.pem" root_ca_cert
#define ximport "certs/my_client.pem" my_client_cert

```

you can provide the rest of the configuration in the Project Defines. Here is an example—copy and paste into the Defines window and modify as necessary:

```

TCPCONFIG = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK = "255.255.255.0";
MY_GATEWAY = "10.10.6.1";
MY_NAMESERVER = "10.10.6.1";
WIFI_USE_WPA;
WIFI_AES_ENABLED;
IFC_WIFI_SSID = "rabbitTest";
IFC_WIFI_ROAM_ENABLE = 1;
IFC_WIFI_ROAM_BEACON_MISS = 20;
IFC_WIFI_MODE = IFPARAM_WIFI_INFRASTRUCTURE;
IFC_WIFI_REGION = IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION = IFPARAM_WIFI_ENCR_CCMP;
WPA_USE_EAP = WPA_USE_EAP_TLS;
IFC_WIFI_WPA_PROTOCOL = IFPARAM_WIFI_WPA_PROTOCOL_WPA2;
IFC_WIFI_CLIENT_CERT_XIM = my_client_cert, my_client_cert;
IFC_WIFI_CA_CERT_XIM = root_ca_cert;
IFC_WIFI_IDENTITY = "MyClientIdent";
SSL_USE_AES;

```

2.4.1.7 Infrastructure, WPA Enterprise using PEAP and TKIP Encryption

To configure within your program, copy and paste the following into your sample program and modify as necessary for your situation:

```
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP "10.10.6.100"
#define _PRIMARY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"
#define WIFI_USE_WPA
#define IFC_WIFI_SSID "rabbitTest"
#define IFC_WIFI_ROAM_ENABLE 1
#define IFC_WIFI_ROAM_BEACON_MISS 20
#define IFC_WIFI_MODE IFPARAM_WIFI_INFRASTRUCTURE
#define IFC_WIFI_REGION IFPARAM_WIFI_REGION_AMERICAS
#define IFC_WIFI_ENCRYPTION IFPARAM_WIFI_ENCR_TKIP
// The above defines are identical to using TKIP with PSK

#define WPA_USE_EAP WPA_USE_EAP_PEAP
#define ximport "certs/root.pem" root_ca_cert
#define IFC_WIFI_CA_CERT_XIM root_ca_cert
#define IFC_WIFI_IDENTITY "my_user1d"
#define IFC_WIFI_PASSWORD "my_passw0rd"
```

Because it is necessary to ximport SSL certificates for use with this encryption method, it is not possible to completely configure WPA Enterprise with PEAP using the Project Defines. However, as long as you include something similar to this #ximport statement in your program:

```
#ximport "certs/root.pem" root_ca_cert
```

you can provide the rest of the configuration in the Project Defines. Here is an example—copy and paste into the Defines window and modify as necessary:

```
TCPCONFIG = 1;
_PRIMARY_STATIC_IP = "10.10.6.100";
_PRIMARY_NETMASK = "255.255.255.0";
MY_GATEWAY = "10.10.6.1";
MY_NAMESERVER = "10.10.6.1";
WIFI_USE_WPA;
IFC_WIFI_SSID = "rabbitTest";
IFC_WIFI_ROAM_ENABLE = 1;
IFC_WIFI_ROAM_BEACON_MISS = 20;
IFC_WIFI_MODE = IFPARAM_WIFI_INFRASTRUCTURE;
IFC_WIFI_REGION = IFPARAM_WIFI_REGION_AMERICAS;
IFC_WIFI_ENCRYPTION = IFPARAM_WIFI_ENCR_TKIP;
WPA_USE_EAP = WPA_USE_EAP_PEAP;
IFC_WIFI_IDENTITY = "my_user1d";
IFC_WIFI_PASSWORD = "my_passw0rd";
IFC_WIFI_CA_CERT_XIM = root_ca_cert;
```

2.4.1.8 Specifying a Pre-Shared Key

There are two ways to specify a pre-shared key—either by specifying the 256-bit pre-shared key directly in hex format, or by entering a passphrase. A passphrase is the most common method. All Wi-Fi devices that accept a passphrase use a standard algorithm to generate the pre-shared key from the passphrase and the SSID. That is, all Wi-Fi devices (including Access Points) that are configured with the same passphrase and the same SSID will generate the same pre-shared key.

IMPORTANT: When statically configuring a Rabbit Wi-Fi device, you should specify either the pre-shared key with `_WIFI_PSK_HEX` or the passphrase with `_WIFI_PSK_PASSPHRASE`, but **NOT BOTH!** If you do specify both, then `_WIFI_PSK_HEX` will take precedence; that is, if you have a correct passphrase but an incorrect pre-shared key, then the incorrect pre-shared key will be used.

2.4.1.9 Ad-hoc, WPA/TKIP or WPA2/CCMP Encryption Configuration

The Wi-Fi driver does not support the use of ad-hoc mode with WPA/TKIP or WPA2/CCMP encryption. Ad-hoc mode with encryption is not well supported by Wi-Fi devices in general. There are multiple standards for supporting encryption on ad-hoc mode, but none of them are widely implemented. If you desire encryption for your Wi-Fi network, then we strongly recommend using infrastructure mode with one or more access points.

2.4.2 Wi-Fi Runtime Configuration

There are two runtime functions for Wi-Fi configuration.

2.4.2.1 Runtime Configuration Starting with Dynamic C 10.40

Dynamic C 10.40 introduces an expanded `ifconfig()` function that includes Wi-Fi configuration parameters. All Wi-Fi programs should switch to using `ifconfig()` for configuring Wi-Fi at runtime.

Dynamic C 10.54 introduces `ifconfig()` parameter identifiers for Wi-Fi Enterprise mode authentication.

2.5 Setting Up PPP Interfaces

PPP interfaces are slightly more complicated to configure than non-PPPoE Ethernet. They also generally take more time to become established. The advantage of PPP is that it can be made to run over a wide variety of physical layer hardware: on Rabbit-based boards this includes the asynchronous serial ports, as well as Ethernet (using PPPoE). Use of PPP over asynchronous serial allows boards with no Ethernet hardware to communicate using TCP/IP protocols.

Starting with Dynamic C version 7.30, the process of establishing a PPP link has been more tightly integrated into the library (using the `ifup/ifdown/ifconfig` functions). Prior to 7.30, your application had to be hard-coded to use either Ethernet, PPP or PPPoE.

“PPP Driver” on page 83 explains the details of establishing PPP interfaces. The following sections provide an overview.

2.5.1 PPP over Asynchronous Serial

There are two basic scenarios for use of PPP over asynchronous serial (shortened here to just PPP). The first is a direct, hard-wired, connection to another machine. The second is a connection to an ISP (Internet

Service Provider) via a modem. Modem connections introduce another layer of complexity in that the modem itself must be instructed to connect to the desired peer's modem, most often via the PSTN (Public Switched Telephone Network). Most often, ISPs also have special requirements for establishing PPP links which are often unrelated to PPP itself. For example, many ISPs require navigation of "login scripts" which are basically intended for human users.

With hard-wired connections, e.g., RS232 cables with "null modems" or "crossed-over connections," the process of establishing a PPP link is relatively simple and reliable. Bringing such a PPP link up involves opening the serial port, sending and receiving PPP link negotiation messages (known as LCP; Link Control Protocol), sending and receiving authentication messages (PAP; Password Authentication Protocol) then finally sending and receiving Internet Protocol Control Messages (IPCP). If all negotiations are successful, the link is then ready for TCP/IP traffic.

If the link is established via a modem, then an extra layer of activity must precede the initial PPP negotiation. This is outside the scope of PPP, since it is really related to the establishment of a physical layer. The TCP/IP library gives you the option of incorporating the modem connection phase into the process of bringing the interface up and down. If preferred, the modem phase can be performed entirely separately from the `ifup()`/`ifdown()` process. This may be necessary if there are special requirements for connecting to the ISP.

2.5.2 PPP over Ethernet

PPPoE is often considered a hack. It seems superfluous to define a protocol that establishes a logical "connection" between two peers on what is otherwise a broadcast (i.e., any-to-any) medium. Nevertheless, the existence of PPPoE was largely dictated by the needs of ISPs who wished to continue using their existing infrastructure, based on the earlier generation of dial-in connections. The advent of high speed (ADSL etc.) modems, that had an Ethernet connection to the user's network, made PPPoE an attractive proposition. If your application requires connection to an ISP via an ADSL modem, then you will most likely need to support PPPoE.

PPPoE also requires a physical layer negotiation to precede the normal PPP negotiations. This is known as the "access concentrator discovery" phase ("discovery" for short). PPPoE makes a distinction between PPPoE servers and PPPoE clients, however, PPP makes no distinction; you can think of PPP as also standing for Peer to Peer Protocol. The PPPoE server is known as the access concentrator. The Dynamic C TCP/IP libraries do not support acting as the access concentrator; only the PPPoE client mode is supported. This is the most common case, since the DSL modem is always configured as an access concentrator.

2.6 Configuration Macro Reference

This section arranges the configuration macros according to the function they perform.

2.6.1 Removing Unnecessary Functions

The following macros default to being undefined (i.e., the functionality is included by default). You can define one or more of these macros to free up code and data memory space.

DISABLE_DNS

This macro disables DNS lookup. This prevents a UDP socket for DNS from being allocated,

thus saving memory. Users may still call `resolve()` with an IP address, provided that the address is in dotted decimal form i.e., does not require a real DNS lookup.

DISABLE_UDP

This macro disables all UDP functionality, including DNS, SNMP, TFTP and DHCP/BOOTP. You can define this to save a small amount of code if your application only needs to be a TCP server, or a TCP client that does not need to do name lookups. This macro is available starting with Dynamic C 7.30.

DISABLE_TCP

This macro disables all TCP functionality, including HTTP (web server), SMTP (mail) and other TCP-based protocols. You can define this to save a substantial amount of code if your application only needs UDP. This macro is available starting with Dynamic C 7.30.

2.6.2 Including Additional Functions

The following macros default to being undefined i.e., the functionality is not included by default.

USE_DHCP

This macro is required when DHCP or BOOTP functionality is desired.

USE_SNMP

Define this to be the version number of SNMP (Simple Network Management Protocol) to be supported. Currently, the only allowable value is '1'.

USE_MULTICAST

This macro will enable multicast support. In particular, the extra checks necessary for accepting multicast datagrams will be enabled and joining and leaving multicast groups (and informing the Ethernet hardware about it) will be added.

USE_IGMP

If this macro is defined, the `USE_MULTICAST` macro is automatically defined. This macro enables sending reports on joining multicast addresses and responding to IGMP queries by multicast routers. Unlike `USE_MULTICAST`, this macro must be defined to be 1 or 2. This indicates which version of IGMP will be supported. Note, however, that both version 1 and 2 IGMP clients will work with both version 1 and 2 IGMP routers. Most users should just choose version 2.

USE_LINKLOCAL

This macro enables the use of link-local addresses (169.254.1.0 to 169.254.254.255) for zero-configuration networking. With `USE_LINKLOCAL` defined, any time you set an Ethernet or Wi-Fi interface's IP address to something on the link-local subnet (169.254.x.x), the TCP/IP stack will automatically select an available address and handle address conflicts with other devices on the network. Link-local addressing can be used with a static IP configuration, or as the fallback address when a DHCP server is unavailable.

2.6.3 BOOTP/DHCP Control Macros

Various macros control the use of DHCP. Apart from setting these macros before `'#use dcrtcp.lib'`, there is typically very little additional work that needs to be done to use DHCP/BOOTP services. Most of the work is done automatically when you call `sock_init()` to initialize TCP/IP. There are more control macros

available than what are listed here. Please look at the beginning of the file `lib\tcpip\bootp.lib` for more information.

USE_DHCP

If this macro is defined, the target uses BOOTP and/or DHCP to configure the required parameters. This macro *must* be defined to use DHCP services.

DHCP_CHECK

If defined, and USE_DHCP is defined, then the target will check for the existence of another host already using an offered IP address, using ARP. If the host exists, then the offer will be declined. If this happened most DHCP servers would log a message to the administrator, since it may represent a misconfiguration. If not defined, then the target will request the first offered address without checking.

DHCP_CLASS_ID "Rabbit2000-TCPIP:Rabbit:Test:1.0.0"

This macro defines a class identifier by which the OEM can identify the type of configuration parameters expected. DHCP servers can use this information to direct the target to the appropriate configuration file. The standard format: "hardware:vendor:product code:firmware version" is recommended.

DHCP_CLIENT_ID clientid_char_ptr

DHCP_CLIENT_ID_LEN clientid_length

Define a client identifier string. Since the client ID can contain binary data, the length of this string must be specified as well. This string **MUST** be unique amongst all clients in an administrative domain, thus in practice the client ID must be individually set for each client e.g., via front-panel configuration. It is **NOT** recommended to program a hard-coded string (as for class ID). Note that RFC2132 recommends that the first byte of the string should be zero if the client ID is not actually the hardware type and address of the client (see next).

DHCP_CLIENT_ID_MAC

If defined, this overrides DHCP_CLIENT_ID, and automatically sets the client ID string to be the hardware type (1 for Ethernet) and MAC address, as suggested by RFC2132.

2.6.4 Buffer and Resource Sizing

MAX_SOCKETS (deprecated)

This macro defines the number of sockets that will be allocated, not including the socket for DNS lookups. It defaults to 4. If libraries such as HTTP.LIB or FTP_SERVER.LIB are used, you must provide enough sockets in MAX_SOCKETS for them also. This macro has been replaced by MAX_TCP_SOCKET_BUFFERS and MAX_UDP_SOCKET_BUFFERS.

MAX_SOCKET_LOCKS

For μ C/OS-II support. This macro defines the number of socket locks to allocate. It defaults to MAX_TCP_SOCKET_BUFFERS + MAX_UDP_SOCKET_BUFFERS.

This macro is necessary because we can no longer calculate the number of socket locks needed based on the number of socket buffers, now that the user can manage their own socket buffers.

MAX_TCP_SOCKET_BUFFERS

Starting with Dynamic C version 7.05, this macro determines the maximum number of TCP sockets with preallocated buffers. If MAX_SOCKETS is defined, then MAX_TCP_SOCKET_BUFFERS will be

assigned the value of `MAX_SOCKETS` for backwards compatibility. If neither macro is defined, `MAX_TCP_SOCKET_BUFFERS` defaults to 4.

MAX_UDP_SOCKET_BUFFERS

Starting with Dynamic C version 7.05, this macro determines the maximum number of UDP sockets with preallocated buffers. It defaults to 0.

SOCK_BUF_SIZE (deprecated)

This macro determines the size of the socket buffers. A TCP socket will have two buffers of size `SOCK_BUF_SIZE/2` for send and receive. A UDP socket will have a single buffer of size `SOCK_BUF_SIZE`. Both types of sockets take the same total amount of buffer space. This macro has been replaced by `TCP_BUF_SIZE` and `UDP_BUF_SIZE`.

TCP_BUF_SIZE

Starting with Dynamic C 7.05, TCP and UDP socket buffers are sized separately. `TCP_BUF_SIZE` defines the buffer sizes for TCP sockets. It defaults to 4096 bytes. Backwards compatibility exists with earlier version of Dynamic C: if `SOCK_BUF_SIZE` is defined, `TCP_BUF_SIZE` is assigned the value of `SOCK_BUF_SIZE`. If `SOCK_BUF_SIZE` is not defined, but `tcp_MaxBufSize` is, then `TCP_BUF_SIZE` will be assigned the value of `tcp_MaxBufSize*2`.

tcp_MaxBufSize (deprecated)

This use of this macro is deprecated in Dynamic C version 6.57 and higher; it has been replaced by `SOCK_BUF_SIZE`.

In Dynamic C versions 6.56 and earlier, `tcp_MaxBufSize` determines the size of the input and output buffers for TCP and UDP sockets. The `sizeof(tcp_Socket)` will be about 200 bytes more than double `tcp_MaxBufSize`. The optimum value for local Ethernet connections is greater than the Maximum Segment Size (MSS). The MSS is 1460 bytes. You may want to lower `tcp_MaxBufSize`, which defaults to 2048 bytes, to reduce RAM usage. It can be reduced to as little as 600 bytes.

`tcp_MaxBufSize` will work slightly differently in Dynamic C versions 6.57 and higher. In these later versions the buffer for the UDP socket will be `tcp_MaxBufSize*2`, which is twice as large as before.

UDP_BUF_SIZE

Starting with Dynamic C 7.05, TCP and UDP socket buffers are sized separately. `UDP_BUF_SIZE` defines the buffer sizes for UDP sockets. It defaults to 4096 bytes. Backwards compatibility exists with earlier version of Dynamic C: if `SOCK_BUF_SIZE` is defined, `UDP_BUF_SIZE` is assigned the value of `SOCK_BUF_SIZE`. If `SOCK_BUF_SIZE` is not defined, but `tcp_MaxBufSize` is, then `UDP_BUF_SIZE` will be assigned the value of `tcp_MaxBufSize*2`.

ETH_MTU

Define the Maximum Transmission Unit for Ethernet and PPPoE interfaces. The default is 600, but may be increased to a maximum of 1500 subject to root data memory limitations. PPPoE always uses a value that is 8 less than this figure. For maximum throughput on an Ethernet link, use the largest value (1500).

Note that, in DC version 7.30, a macro will be defined which is set to the larger of `ETH_MTU` and `PPP_MTU`. This macro is called `MAX_MTU`, and is used for sizing the receive buffer for incoming packets from all interfaces.

PPP_MTU

Define the maximum transmission/receive unit for PPP over serial links. This defaults to the same as ETH_MTU if it is defined, or 600. This macro is new for 7.30.

ETH_MAXBUFS

Define the maximum number of incoming packets that may be buffered. Defaults to 10. The buffers are shared between all interfaces (in spite of the name). The total amount of root data storage for incoming packets depends on the configured mix of interface types, but is $(MAX_MTU+22)*ETH_MAXBUFS$ for just Ethernet without PPPoE. This will default to 6220 bytes if the defaults are selected.

ARP_TABLE_SIZE

Define to the number of ARP table entries. The default is set to the number of interfaces, plus 5 entries for every Ethernet interface (excluding PPPoE). The maximum allowable value is 200.

ARP_ROUTER_TABLE_SIZE

Define the maximum number of routers. Defaults to the number of interfaces, plus an extra entry for each Ethernet (excluding PPPoE) .

MAX_STRING

Define the maximum number of characters for a hostname or for a mail server when using the function `smtp_setserver ()`. Defaults to 50.

MAX_NAMESERVERS

Define the maximum number of DNS servers. Defaults to 2.

MAX_COOKIES

Define the maximum number of cookies that a server can send to or receive from a client. Defaults to 1.

TCP_MAXPENDING

Define the maximum number of pending TCP connections allowed in the active list. Defaults to 20.

MAX_RESERVEPORTS

Defines the maximum number of TCP port numbers that may be reserved. Defaults to 5 if USE_RESERVEDPORTS is defined (which is defined by default). For more information about USE_RESERVEDPORTS and setting up a listen queue, please see “Specifying a Listen Queue” on page 46.

DNS_MAX_RESOLVES

4 by default. This is the maximum number of concurrent DNS queries. It specifies the size of an internal table that is allocated in xmem.

DNS_MAX_NAME

64 by default. Specifies the maximum size in bytes of a host name that can be resolved. This number includes any appended default domain and the NULL-terminator. Backwards compatibility exists for the MAX_DOMAIN_LENGTH macro. Its value will be overridden with the value DNS_MAX_NAME if it is defined.

For temporary storage, a variable of this size must be placed on the stack in DNS processing. Normally, this is not a problem. However, for μ C/OS-II with a small stack and a large value for `DNS_MAX_NAME`, this could be an issue.

DNS_MAX_DATAGRAM_SIZE

512 by default. Specifies the maximum length in bytes of a DNS datagram that can be sent or received. A root data buffer of this size is allocated for DNS support.

DNS SOCK_BUF_SIZE

1024 by default. Specifies the size in bytes of an xmem buffer for the DNS socket. Note that this means that the DNS socket does not use a buffer from the socket buffer pool.

2.6.5 Network Configuration Prior to Dynamic C 7.30

These macros should only be used for releases of Dynamic C prior to version 7.30. They are supported in newer releases of Dynamic C for backward compatibility; however, new applications should use the new style of configuration outlined in “Network Configuration Starting with Dynamic C 7.30” on page 31. Use of the runtime functions mentioned in this section is deprecated in favor of `ifconfig()`.

MY_DOMAIN

This macro is the initial value for the domain portion of the controller’s address. At runtime, it can be overwritten by `tcp_config()` and `setdomainname()`.

MAX_DOMAIN_LENGTH

Specify the maximum domain name length, including any concatenated host name. Defaults to 128.

MY_GATEWAY

This macro gives the default value for the controllers default gateway. At runtime, it can be overwritten by `tcp_config()`.

MY_IP_ADDRESS

This macro is the default IP address for the controller. At runtime, it can be overwritten by `tcp_config()` and `sethostid()`.

MY_NAMESERVER

This macro is the default value for the primary name server. At runtime, it can be overwritten by `tcp_config()`.

MY_NETMASK

This macro is the default netmask for the controller. At runtime, it can be overwritten by `tcp_config()`.

2.6.6 Network Configuration Starting with Dynamic C 7.30

These macros should only be used with Dynamic C 7.30 or later.

TCPCONFIG

Define to the number of a predefined configuration in `tcp_config.lib` (numbers less than 100) or `custom_config.lib` (numbers greater or equal to 100). Defaults to 0, which means no predefined configuration.

USE_ETHERNET

Define to 0 (or leave undefined) if Ethernet is not required. Define to 1 if the first Ethernet port is to be used. Defaults to 0. This macro does not include PPPoE interfaces.

USE_PPP_SERIAL

Define to a bitwise-OR combination of:

- SERA (0x01) - Serial port A (IF_PPP0)
- SERB (0x02) - Serial port B (IF_PPP1)
- SERC (0x04) - Serial port C (IF_PPP2)
- SERD (0x08) - Serial port D (IF_PPP3)
- SERE (0x10) - Serial port E (IF_PPP4)
- SERF (0x20) - Serial port F (IF_PPP5)

Defaults to 0, i.e., no PPP over serial.

USE_PPPOE

Define in the same way as USE_ETHERNET, except that PPPoE is used on the specified Ethernet port. Defaults to 0 i.e., no PPPoE interfaces.

USE_WIFI

Define to 1 if Wi-Fi is required. Defaults to 0.

IFCONFIG_ALL

IFCONFIG_DEFAULT

IFCONFIG_ETH0

IFCONFIG_PPP0..5

IFCONFIG_PPPOE0

IFCONFIG_WIFI0

All the above IFCONFIG_* macros are defined in a similar manner. IFCONFIG_ALL is reserved for configuration items that are not specific to any particular interface number. IFCONFIG_DEFAULT is applied to the default interface (IF_DEFAULT) if there is no specific IFCONFIG_* for the default interface.

These macros must be defined as a C parameter list fragment. This is because the macro value is substituted into a call to `ifconfig()` at initialization time (`sock_init()`). For example, the fragment of code that initializes the non-PPPoE Ethernet interface looks somewhat like the following:

```
#ifdef IF_ETH0
    #ifdef IFCONFIG_ETH0
        ifconfig(IF_ETH0, IFCONFIG_ETH0, IFS_END);
    #else
        #if IF_DEFAULT == IF_ETH0
            ifconfig(IF_DEFAULT, IFCONFIG_DEFAULT, IFS_END);
        #endif
    #endif
#endif
```

The entire fragment is processed only if IF_ETH0 is defined, i.e., you have specified that the non-PPPoE Ethernet interface is to be used. Inside this, if the IFCONFIG_ETH0 macro has been defined, then it is

substituted into an `ifconfig()` call for `IF_ETH0`. Otherwise, if `IF_ETH0` is the default (i.e., equal to `IF_DEFAULT`) then the `IFCONFIG_DEFAULT` macro is substituted into the `ifconfig()` call.

Note that for backwards compatibility, `IFCONFIG_DEFAULT` is always defined to something if it was not explicitly defined prior to inclusion of `dcrtcp.lib`. It is defined using the given values of the pre version 7.30 macros: `MY_IP_ADDRESS`, `MY_GATEWAY` etc.

The `IFCONFIG_*` macros can be defined to be an arbitrary number of `ifconfig()` parameters. For example,

```
#define IFCONFIG_ETH0\  
    IFS_IPADDR, aton("10.10.6.100"), \  
    IFS_NETMASK, 0xFFFFFFFF00uL, \  
    IFS_ROUTER_ADD, aton("10.10.6.1"), \  
    IFS_ROUTER_ADD_STATIC, aton("10.10.6.111"), \  
        aton("10.10.6.0"), 0xFFFFFFFF00uL, \  
    IFS_DEBUG, 5, \  
    IFS_ICMP_CONFIG, 1, \  
    IFS_UP
```

sets up local IP address and netmask, two routers, turns the verbose level all the way up, allows ping configure, and finally specifies that the interface be brought up at boot time.

The final `IFS_UP` is important: if it is omitted, then the interface will not be brought up at boot time; you will need to call `ifup()` explicitly after `sock_init()`.

For a full list of the parameters that you can specify in an `IFCONFIG_*` macro, please see the documentation for `ifconfig()`.

2.6.7 Time-Outs and Retry Counters

RETRAN_STRAT_TIME

This is used for several purposes. It is the minimum time granularity (in milliseconds) of the retransmit process. No time-out is set less than this value. It defaults to 10 ms.

TCP_OPENTIMEOUT

Defines the time-out value (in milliseconds) for active open processing. Defaults to 31000 ms.

TCP_CONNTIMEOUT

Defines the time-out value in milliseconds during open or close negotiation. Defaults to 13000 ms.

TCP_SYNQTIMEOUT

Defines the time-out value (in milliseconds) for pending connection. Defaults to 90000 ms.

TCP_TWTIMEOUT

Define time to linger in `TIMEWAIT` state (milliseconds). It should be from .5 to 4 minutes (2MSL) but it's not really practical for us. Two seconds will hopefully handle the case where ACK must be retransmitted, but can't protect future connections on the same port from old packets. Defaults to 2000 ms.

KEEPALIVE_NUMRETRYS

Number of times to retry the TCP keepalive. Defaults to 4.

KEEPALIVE_WAITTIME

Time (in seconds) to wait for the response to a TCP keepalive. Defaults to 60 seconds.

TCP_MAXRTO

Set an overall upper bound for the retransmit timeout, in milliseconds. Defaults to 50,000 ms.

TCP_MINRTO

Set a lower bound for the retransmit timeout, in units of milliseconds. Prior to Dynamic C 9.01, the default value for this macro was 250 ms (¼ second). Starting with Dynamic C 9.01, stack improvements allowed the default value to be reduced to 10 ms. In either case, beware of reducing this value, since modern hosts try to ack only every second segment. If our RTO is too small, we will unnecessarily retransmit if we don't get the ack for the first of the two segments (especially on a fast LAN, where the RTT measurement will want to make us set a small time-out).

TCP_LAZYUPD

Set a delay time for "lazy update" (ms). This is used to slightly delay window updates and empty acknowledgments to the peer, in the hope of being able to tag extra data along with otherwise empty segments. This improves performance by allowing better interleaving of application processing with TCP activity, and sending fewer empty segments. This delay interval is also used when we need to retransmit owing to a temporary shortage of Ethernet transmit buffers. Defaults to 5 ms.

DNS_RETRY_TIMEOUT

2000 by default. Specifies the number of milliseconds to wait before retrying a DNS request. If a request to a nameserver times out, then the next nameserver is tried. If that times out, then the next one is tried, in order, until it wraps around to the first nameserver again (or runs out of retries).

DNS_NUMBER_RETRIES

2 by default. Specifies the number of times a request will be retried after an error or a timeout. The first attempt does not constitute a retry. A retry only occurs when a request has timed out, or when a nameserver returns an unintelligible response. That is, if a host name is looked up and the nameserver reports that it does not exist and then the DNS resolver tries the same host name with or without the default domain, that does not constitute a retry.

DNS_MIN_KEEP_COMPLETED

10000 by default. Specifies the number of milliseconds a completed request is guaranteed to be valid for `resolve_name_check()`. After this time, the entry in the internal table corresponding to this request can be reused for a subsequent request.

2.6.8 Program Debugging

TCP_STATS

Enable TCP socket statistics collection. This causes some additional fields to be defined in the TCP socket structure, which are updated with various counters. This is mainly for internal debugging.

DCRTCP_DEBUG

If defined, allow Dynamic C debugging in all TCP/IP libraries. This allows you to trace into library functions in case you are finding difficulty in solving a TCP/IP problem. Remember to remove this definition when compiling for a production environment.

DCRTCP_VERBOSE

If defined, enable debugging messages to be printed by the library to the Dynamic C stdout window. This can be very informative when you are trying to see how the TCP/IP libraries work. Unfortunately, the string messages take up a lot of root code space, so you may need to increase the DATAORG value in the BIOS. Otherwise, you can be more selective about which messages are printed by defining *_VERBOSE macros for individual libraries (DCRTCP_VERBOSE merely turns on all the individual library verbose definitions). See dcrtcp.lib source for a listing of the available debug and verbose macros.

Note that the number of messages printed depends on the value of a global variable, debug_on. If this variable is 0, only a few messages are printed. If set to higher numbers (up to 5), then successively more detailed messages are printed. You can set this variable directly at the start of your main() function, or preferably use

```
ifconfig(IF_ANY, IFS_DEBUG, 5, IFS_END);
```

2.6.9 Miscellaneous Macros

TCP_FASTSOCKETS

Define to '1' if sockets connected to "reserved" ports can be closed without the usual 2MSL delay. The default is set to '1', define to '0' to override this.

NET_ADD_ENTROPY

Define this macro to allow network packet arrival times (from any interface) to be a source of random number seeds. See RAND.LIB for further information.

NET_COARSELOCK

This macro is only used when μ C/OS-II is active. It affects the definition of 2 other macros:

LOCK_SOCKET(s) and UNLOCK_SOCKET(s).

If NET_COARSELOCK is not defined, the lock/unlock macros are individual socket locks for use on socket transmit/receive buffers and the socket structure itself. If it is defined, the lock/unlock macros are global locks.

TCP_NO_CLOSE_ON_LAST_READ

If defined, then support half-close; i.e., sock_close() only closes the transmit side of the socket, but allows indefinite receives until the peer closes. This prevents the normal close timeout from being set. Also, when reading, if the socket is half-closed by the peer, then the socket will be automatically closed from this side if this define is *not* set.

2.6.9.1 TOS and TTL

TOS and TTL are fields in the IP header. TOS, short for "Type of Service," uses 4 bits to specify different types of service. For normal service all 4 bits are zero. Different applications will want different types of service. For example, SNMP might set the maximize reliability bit, whereas FTP would want maximize throughput.

- IPTOS_DEFAULT is normal service.
- IPTOS_CHEAP minimizes monetary cost.
- IPTOS_RELIABLE maximizes reliability.
- IPTOS_CAPACIOUS maximizes throughput

- `IPTOS_FAST` minimizes delay.
- `IPTOS_SECURE` maximizes security.

Note that you may not OR these values together. You must pick one only!

TTL (Time to Live) specifies how many routers a packet may visit before it is discarded, or how many seconds it can remain in the network, whichever comes first.

TCP_TTL

Default TTL of TCP segments. This value is from Internet STD0002. Defaults to 64.

TCP_TOS

Default type of service for TCP. Defaults to `IPTOS_DEFAULT`.

UDP_TTL

Default TTL of UDP datagrams. This value is from Internet STD0002. Defaults to 64.

UDP_TOS

Default type of service for UDP. Defaults to `IPTOS_DEFAULT`.

ICMP_TOS

Default type of service for ICMP. Defaults to `IPTOS_DEFAULT`.

2.6.10 Wi-Fi Configuration Macros

To configure a Rabbit device for a Wi-Fi network connection, you will need to define a number of macros. As shown in “Wi-Fi Compile Time Configuration” on page 17, these can be defined in either the C program or the Defines window.

An important point to remember is that the configuration of TCP/IP applies equally well to Wi-Fi devices as it does to Ethernet devices.

TCPCONFIG

This macro works the same for a Wi-Fi interface as it does for an Ethernet interface. Use the value 1 for a static TCP/IP network configuration, or 5 for a DHCP (dynamic) configuration.

IFC_WIFI_SSID (default "rabbitTest")

This is the SSID (Service Set Identifier) or name of the wireless network that you want your Rabbit device to connect to. All other devices on your wireless network (including your access point or wireless router) must have this same name.

(The macro `_WIFI_SSID` was deprecated in Dynamic C 10.40.)

IFC_WIFI_ROAM_ENABLE (default 1)

This turns roaming on or off.

IFC_WIFI_ROAM_BEACON_MISS (default 20)

This sets the number of beacons that are missed continuously in order for scanning for a better access point and subsequent association to take place.

IFC_WIFI_MODE (default IFPARAM_WIFI_INFRASTRUCTURE)

Specifies the network architecture mode for your wireless network. This macro has two possible values. `IFPARAM_WIFI_INFR` means that the device will attach to an infrastructure network. An infrastructure network includes at least one access point that coordinates communication on the wireless network. `IFPARAM_WIFI_ADHOC` means that the device will use an ad-hoc mode network. An ad-hoc network does not have an access point; all devices on the wireless network simply communicate directly with each other.

(The macro `_WIFI_MODE` was deprecated in Dynamic C 10.40.)

IFC_WIFI_CHANNEL

Sets the channel (1-14) to use for an ad-hoc network.

(The macro `_WIFI_OWNCCHANNEL` was deprecated in Dynamic C 10.40.)

IFC_WIFI_REGION (default IFPARAM_WIFI_REGION_AMERICAS)

This macro sets the channel range and maximum power limit for the region selected. Different regions have different regulations on Wi-Fi communication. This macro can have the value:

- `IFPARAM_WIFI_REGION_AMERICAS` - Americas, including the US (ch. 1-11)
- `IFPARAM_WIFI_REGION_AUSTRALIA` - Australia (ch. 1-11)
- `IFPARAM_WIFI_REGION_CANADA` - Canada (ch. 1-11)
- `IFPARAM_WIFI_REGION_CHINA` - China (ch. 1-11)
- `IFPARAM_WIFI_REGION_EMEA` - Europe, Middle East, Africa (ch. 1-13)
- `IFPARAM_WIFI_REGION_FRANCE` - France (ch. 10-13)
- `IFPARAM_WIFI_REGION_ISRAEL` - Israel (ch. 3-11)
- `IFPARAM_WIFI_REGION_JAPAN` - Japan (ch. 1-13)
- `IFPARAM_WIFI_REGION_MEXICO_INDOORS` - Mexico indoors (ch. 1-11)
- `IFPARAM_WIFI_REGION_MEXICO_OUTDOORS` - Mexico outdoors (ch. 9-11)

(The macro `_WIFI_REGION_REQ` was deprecated in Dynamic C 10.40.)

IFC_WIFI_ENCRYPTION (default IFPARAM_WIFI_ENCR_NONE)

This parameter controls the type of encryption used. Select one of the following:

- `IFPARAM_WIFI_ENCR_ANY` - Use any type of encryption.
- `IFPARAM_WIFI_ENCR_NONE` - No encryption used.
- `IFPARAM_WIFI_ENCR_WEP` - Use WEP encryption.
- `IFPARAM_WIFI_ENCR_TKIP` - Use WPA encryption; the macro `WIFI_USE_WPA` must be defined to use this parameter.
- `IFPARAM_WIFI_ENCR_CCMP` - Use WPA2 encryption; both `WIFI_USE_WPA` and `WIFI_AES_ENABLED` must be defined to use this parameter.

(The macro `_WIFI_WEP_FLAG` was deprecated in Dynamic C 10.40.)

IFC_WIFI_WEP_KEYNUM (default 0)

Select the WEP key to use when using WEP encryption. This macro can have the value 0, 1, 2, or 3, which corresponds to the WEP key macros (`IFC_WIFI_WEP_KEY#_*`) described next.

(The macro `_WIFI_USEKEY` was deprecated in Dynamic C 10.40.)

IFC_WIFI_WEP_KEY0_BIN

IFC_WIFI_WEP_KEY1_BIN
IFC_WIFI_WEP_KEY2_BIN
IFC_WIFI_WEP_KEY3_BIN

Specifies the possible WEP keys to use for WEP encryption. They default to undefined.

These keys can be either 40-bit or 104-bit (i.e., 5 bytes or 13 bytes) and must be a comma-separated list of byte values. The `IFC_WIFI_WEP_KEY#_BIN` macro that is used depends on the value of `IFC_WIFI_WEP_KEYNUM`.

(The `_WIFI_KEY#` macros were deprecated in Dynamic C 10.40.)

IFC_WIFI_WEP_KEY0_HEXSTR
IFC_WIFI_WEP_KEY1_HEXSTR
IFC_WIFI_WEP_KEY2_HEXSTR
IFC_WIFI_WEP_KEY3_HEXSTR

Specifies the possible WEP keys to use for WEP encryption. They default to undefined.

These keys can be either 40-bit or 104-bit (i.e., a string of either 10 or 26 hex characters). Note that you do not necessarily need to define all four WEP keys. You can typically just define one key, make sure it matches the key used on all other devices, and set `IFC_WIFI_WEP_KEYNUM` to point to the correct key.

If both `IFC_WIFI_WEP_KEY#_HEXSTR` and `IFC_WIFI_WEP_KEY#_BIN` are defined for a particular key, the HEX version will be used.

IFC_WIFI_WPA_PSK_PASSPHRASE

TKIP encryption requires a passphrase or a key. This macro allows you to define a passphrase with an ASCII string. The Wi-Fi driver will expand the passphrase into a key using a standard algorithm. This process takes up to 20 seconds. The same passphrase must be configured on all devices on the same Wi-Fi network.

(The macro `_WIFI_PSK_PASSPHRASE` was deprecated in Dynamic C 10.40.)

IFC_WIFI_WPA_PSK_HEXSTR

Instead of a passphrase, you can specify a key directly for TKIP encryption. The key must be given as a series of hex digits within an ASCII string. The key must be 256 bits, or 64 hex digits. Specifying the key directly means that a passphrase does not need to be expanded, thus speeding startup time.

IMPORTANT: When statically configuring a Rabbit Wi-Fi device, specify either the pre-shared key with `IFC_WIFI_WPA_PSK_HEXSTR` or the passphrase with `IFC_WIFI_WPA_PSK_PASSPHRASE`, but NOT BOTH! If you do specify both, then `IFC_WIFI_WPA_PSK_HEXSTR` will take precedence; that is, if you have a correct passphrase but an incorrect pre-shared key, then the incorrect pre-shared key will be used.

(The macro `_WIFI_PSK_HEX` was deprecated in Dynamic C 10.40.)

IFC_WIFI_WPA_PROTOCOL

Set to bitwise OR of the following:

- `IFPARAM_WIFI_WPA_PROTOCOL_WPA2`
- `IFPARAM_WIFI_WPA_PROTOCOL_WPA`

to support WPA2 (RSN (Robust Security Network) with CCMP) or WPA (the original TKIP-based WiFi Protected Access).

IFC_WIFI_CLIENT_CERT_XIM

Set to two parameters: first parameter is public certificate of this (client) machine for EAP/TLS. Second parameter is for the corresponding private key. These are generally #ximported files, which are the .der or .pem certificate/private key files. Currently, the private key file must *not* itself be encrypted. This is only used for EAP-TLS.

Note: if certificates have been generated by the Rabbit Certificate Utility, then you should #ximport the <basename>.pem and <basename>key.pem files, where <basename> is the file name you selected for the client certificate. For example, if the file name was “client” then

```
#ximport "certs/clients.pem" my_client_cert
#ximport "certs/clientkey.pem" my_client_key
#define IFC_WIFI_CLIENT_CERT_XIM my_client_cert, my_client_key
```

IFC_WIFI_CA_CERT_XIM root_ca_cert

Set to the #ximported file containing the trusted certificate authority for validating the Authentication Server's public certificate. This may be a .pem or .der format file.

This is used for EAP-TLS, and should also be used for PEAP since PEAP uses TLS “under the covers.” In either case, it is important to be able to verify the authenticity of the Authentication Server itself. This is done by having the trusted CA certificate available.

Note: if you generated your own Certificate Authority (CA) using the Rabbit Certificate Utility, then you should #ximport the <basename>.der or <basename>.pem file for the CA. For example, if the file name was “my_ca” then:

```
#ximport "private/my_ca.der" root_ca_cert
#define IFC_WIFI_CA_CERT_XIM root_ca_cert
```

IFC_WIFI_IDENTITY

Define an identity for PEAP. Using this macro sets both the inner and outer identity to the same value.

```
#define IFC_WIFI_IDENTITY "MyClientIdent"
```

If your application requires a different outer “plaintext” identity, then your application will need to use the `ifconfig()` function directly.

This macro is also required for EAP-TLS. The userID may be set to a dummy value, depending on the requirements of the Authentication Server.

IFC_WIFI_PASSWORD

Define a password for PEAP. This is not used for EAP-TLS.

```
#define IFC_WIFI_PASSWORD "my_passw0rd"
```

IFC_WIFI_AUTHENTICATION (default IFPARAM_WIFI_AUTH_ANY)

Specifies the authentication mode to use for this Wi-Fi network. It accepts a combination (with multiple values ORed together) of the following values:

- IFPARAM_WIFI_AUTH_ANY - Use any method.

- `IFPARAM_WIFI_AUTH_OPEN` - Use open authentication.
- `IFPARAM_WIFI_AUTH_SHAREDKEY` - Use WEP shared-key authentication.
- `IFPARAM_WIFI_AUTH_WPA_PSK` - Use WPA pre-shared key (TKIP, CCMP); the macro `WIFI_USE_WPA` must be defined to use this parameter. This implies the IEEE 802.11i “Personal” operating mode.
- `IFPARAM_WIFI_AUTH_WPA_8021X` - Use IEEE 802.1X/EAP authentication. This implies the IEEE 802.11i Enterprise operating mode. For more information on Wi-Fi Enterprise, refer to: [An Introduction to Wi-Fi](#), available online and with the Dynamic C distribution.

(The macro `_WIFI_AUTH_MODE` was deprecated in Dynamic C 10.40.)

IFC_WIFI_FRAG_THRESHOLD (default 0)

Set the fragmentation threshold. Frames (or packets) that are larger than this threshold are split into multiple fragments. This can be useful on busy or noisy networks. The range is 256-2346, or 0 which means no fragmentation.

(The macro `_WIFI_FRAG_THRESH` was deprecated in Dynamic C 10.40.)

IFC_WIFI_RTS_THRESHOLD (default 0)

Set the RTS threshold, the frame size at which the RTS/CTS mechanism is used. This is sometimes useful on busy or noisy networks. Its range is 0-2347, where 0 is the default (i.e., no RTS/CTS).

(The macro `_WIFI_RTS_THRESH` was deprecated in Dynamic C 10.40.)

SSL_USE_AES

This is required for EAP/TLS, since it uses SSL internally.

WIFI_USE_WPA

Define this macro to use the WPA functionality in the Wi-Fi driver. This is necessary to enable TKIP encryption. Because the WPA code has a significant size, it is not compiled by default.

WPA_USE_EAP

Set to a bitwise OR combination of the following:

- `WPA_USE_EAP_TLS`
- `WPA_USE_EAP_PEAP`

which specifies the Wi-Fi Enterprise authentication method to use, currently either EAP-TLS or PEAP. Use of this macro will bring in a lot of library code to support Enterprise level authentication and key management. This will require configuration of the access point as well as a separate machine to be used as an Authentication Server (e.g., a RADIUS server).

3. TCP and UDP Socket Interface

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are both transport layer protocols. TCP is used when a reliable, stream-oriented, transport is required for data flowing between two hosts on a network. UDP is a record-oriented protocol which is used when lower overhead is more important than reliability. The acronym UDP is sometimes expanded as “unreliable datagram protocol” although, in practice, UDP is quite reliable especially over a local Ethernet LAN segment.

The Dynamic C TCP/IP libraries implement TCP and UDP over IP (Internet Protocol). IP is a network layer protocol, that in turn uses lower levels known as “link layer” protocols, such as Ethernet and PPP (Point-to-Point Protocol). The link-layer protocols depend on a physical layer, such as 10BaseT for Ethernet, or asynchronous RS232 for PPP over serial.

In the other direction, various protocols use TCP. This includes the familiar protocols HTTP, SMTP (mail) and FTP. Other protocols use UDP: DNS and SNMP to name a couple. TCP handles a lot of messy details which are necessary to ensure reliable data flow in spite of possible deficiencies in the network, such as lost or re-ordered packets. For example, TCP will automatically retransmit data that was not acknowledged by the peer within a reasonable time. TCP also paces data transmission so that it does not overflow the peer’s receive buffers (which are always finite) and does not overload intermediate nodes (routers) in the network. UDP leaves all of these details to the application, however UDP has some benefits that TCP cannot provide: one benefit is that UDP can “broadcast” to more than one peer, and another is that UDP preserves the concept of “record boundaries” which can be useful for some applications.

TCP is a connection-oriented protocol. Two peers establish a TCP connection, which persists for the exclusive use of the two parties until it is mutually closed (in the usual case). UDP is connectionless. There is no special start-up or tear-down required for UDP communications. You can send a UDP packet at any time to any destination. Of course, the destination may not be ready to receive UDP packets, so the application has to handle this possibility. (In spite of being “connectionless,” we still sometimes refer to UDP “connections” or “sessions” with the understanding that the connection is a figment of your application’s imagination.)

This chapter describes how to implement your own application level protocols on top of TCP or UDP. The Dynamic C TCP/IP libraries can also be examined for further hints as to how to code your application. For example, `HTTP.LIB` contains the source for an HTTP web server.

3.1 What is a Socket?

Both TCP and UDP make extensive use of the term “socket.” A TCP socket represents the connection state between the local host and the remote peer. When talking about TCP connections that traverse the Internet, a socket is globally unique because it is described by 4 numbers: the local and remote IP addresses (32 bits each), and the local and remote port numbers (16 bits each).

Connections that do not traverse the Internet (e.g., between two hosts on an isolated LAN) are still unique within the attached network.

UDP sockets do not have the global uniqueness property, since they are not connection-oriented. For UDP, a socket really refers to just the local side.

For practical purposes, a socket is a structure in RAM that contains all the necessary state information. TCP sockets are considerably larger than UDP sockets since there is more connection state information to maintain. TCP sockets also require both a receive and a transmit buffer, whereas UDP sockets require only a receive buffer.

With Dynamic C version 6.57, each socket must have an associated `tcp_Socket` structure of 145 bytes or a `udp_Socket` structure of 62 bytes. The I/O buffers are in extended memory. For Dynamic C 7.30 these sizes are 136 bytes and 44 bytes, respectively.

For earlier versions of Dynamic C (than 6.57), each socket must have a `tcp_Socket` data structure that holds the socket state and I/O buffers. These structures are, by default, around 4200 bytes each. The majority of this space is used by the input and output buffers.

3.1.1 Port Numbers

Both TCP and UDP sockets make use of port numbers. Port numbers are a convenient method of allowing several simultaneous connections to exist between the same two hosts. Port numbers are also used to provide “well-known” starting points for common protocols. For example, TCP port number 23 is used for standard telnet connections. In general, port numbers below 1024 are used for standard services. Numbers between 1024 and 65535 are used for connections of a temporary nature. Often, the originator of a connection will select one of the temporary port numbers for its end of the connection, with the well-known number for the other end (which is often some sort of “server”).

TCP and UDP port numbers are not related and operate in an independent “space.” However, the well-known port numbers for TCP and UDP services often match if the same sort of protocol can be made to run over TCP or UDP.

When you open a socket using the TCP/IP libraries, you can specify a particular port number to use, or you can allow the library to pick a temporary port number for an “ephemeral” connection.

3.2 Allocating TCP and UDP Sockets

In all versions of Dynamic C, TCP and UDP socket structures *must* be allocated in static data storage. This is simply accomplished by declaring a static variable of type `tcp_Socket` or `udp_Socket`:

```
static tcp_Socket my_sock;  
static udp_Socket my_udp_sock_array[20];
```

3.2.1 Allocating Socket Buffers

Starting with Dynamic C version 7.05, there are two macros that define the number of sockets available. These macros do not determine how many sockets you can allocate, but they do limit how many sockets you can successfully use. Each socket requires some resources which are not automatically available just because you declare a `tcp_Socket` structure. The additional resources are receive/transmit buffers (which are allocated in extended memory), and also socket semaphores if you are using μ C-OS/II. The relevant macros are:

MAX_TCP_SOCKET_BUFFERS

Determines the maximum number of TCP sockets with preallocated buffers. The default is 4. A buffer is tied to a socket with the first call to `tcp_open()` or `tcp_listen()`. If you use `tcp_extopen()` or `tcp_extlisten()` then these buffer resources are not used up, but only if you allocate your own buffers using `xalloc()`.

MAX_UDP_SOCKET_BUFFERS

Determines the maximum number of UDP sockets with preallocated buffers. The default is 0. A buffer is tied to a socket with the first call to `udp_open()`. If you use `udp_extopen()` then these buffer resources are not used up, but only if you allocate your own buffers using `xalloc()`.

Note that DNS does not need a UDP socket buffer since it manages its own buffer. Prior to version 7.30, DHCP and `TFTP.LIB` each need one UDP socket buffer. Starting with version 7.30, DHCP manages its own socket buffers.

Prior to Dynamic C version 7.05, `MAX_SOCKETS` defined the number of sockets that could be allocated, not including the socket for DNS lookups. If you use libraries such as `HTTP.LIB` or `FTP_SERVER.LIB`, you must provide enough sockets in `MAX_SOCKETS` for them also.

In Dynamic C 7.05 (and later), if `MAX_SOCKETS` is defined in an application program, `MAX_TCP_SOCKET_BUFFERS` will be assigned the value of `MAX_SOCKETS`.

If you are using μ C-OS/II then there is a further macro which must be set to the correct value: `MAX_SOCKET_LOCKS`. This must count every socket (TCP plus UDP), including those used internally by the libraries. If you cannot calculate this exactly, then it is best to err on the side of caution by overestimating. The actual socket semaphore structure is not all that big (less than 70 bytes).

The default value for `MAX_SOCKET_LOCKS` is the sum of `MAX_TCP_SOCKET_BUFFERS` and `MAX_UDP_SOCKET_BUFFERS` (plus 1 if DNS is being used).

3.2.2 Socket Buffer Sizes

Starting with Dynamic C version 7.05, TCP and UDP I/O buffers are sized separately using:

TCP_BUF_SIZE

Determines the TCP buffer size. Defaults to 4096 bytes.

UDP_BUF_SIZE

Determines the UDP buffer size. Defaults to 4096 bytes.

Compatibility is maintained with earlier versions of Dynamic C. If `SOCK_BUF_SIZE` is defined, `TCP_BUF_SIZE` and `UDP_BUF_SIZE` will be assigned the value of `SOCK_BUF_SIZE`. If `SOCK_BUF_SIZE` is not defined, but `tcp_MaxBufSize` is, then `TCP_BUF_SIZE` and `UDP_BUF_SIZE` will be assigned the value of `tcp_MaxBufSize * 2`.

3.2.2.1 User-Supplied Buffers

Starting with Dynamic C version 7.05, a user can associate his own buffer with a TCP or UDP socket. The memory for the buffer must be allocated by the user. This can be done with `xalloc()`, which returns a pointer to the buffer. This buffer will be tied to a socket by a call to an extended open function: `tcp_extlisten()`, `tcp_extopen()` or `udp_extopen()`. Each function requires a long pointer to the buffer and its length be passed as parameters.

3.3 Opening TCP Sockets

There are two ways to open a TCP socket, passive and active. Passive open means that the socket is made available for connections originated from another host. This type of open is commonly used for Internet servers that listen on a well-known port, like 80 for HTTP (Hypertext Transfer Protocol) servers. Active open is used when the controller board is establishing a connection with another host which is (hopefully) listening on the specified port. This is typically used when the controller board is to be a “client” for some other server.

The distinction between passive and active open is lost as soon as the connection is fully established. When the connection is established, both hosts operate on a peer-to-peer basis. The distinction between who is “client” and who is “server” is entirely up to the application. TCP itself does not make a distinction.

3.3.1 Passive Open

To passively open a socket, call `tcp_listen()` or `tcp_extlisten()`; then wait for someone to contact your device. You supply the listen function with a pointer to a `tcp_Socket` data structure, the local port number others will be contacting on your device, and possibly the IP address and port number that will be acceptable for the peer. If you want to be able to accept connections from any IP address or any port number, set one or both to zero.

To handle multiple simultaneous connections, each new connection will require its own `tcp_Socket` and a separate call to one of the listen functions, but using the same local port number (`lport` value). The listen function will immediately return, and you must poll for the incoming connection. You can manually poll the socket using `sock_established()`. The proper procedure for fielding incoming connections is described below.

3.3.2 Active Open

When your Web browser retrieves a page, it actively opens one or more connections to the server's passively opened sockets. To actively open a connection, call `tcp_open()` or `tcp_extopen()`, which use parameters that are similar to the ones used in the listen functions. Supply exact parameters for `remip` and `port`, which are the IP address and port number you want to connect to; the `lport` parameter can be zero, causing an unused local port between 1024 and 65535 to be selected.

If the open function returns zero, no connection was made. This could be due to routing difficulties, such as an inability to resolve the remote computer's hardware address with ARP. Even if non-zero is returned, the connection will not be immediately established. You will need to check the socket status as described in the next section.

3.3.3 Waiting for Connection Establishment

When you open a TCP socket either passively or actively, you must wait for a complete TCP connection to be established. This is technically known as the "3-way handshake." As the name implies, at least 3 packets must be exchanged between the peers. Only after completion of this process, which takes at least one round-trip time, does the connection become fully established such that application data transfer can proceed.

Unfortunately, the 3-way handshake may not always succeed: the network may get disconnected; the peer may cancel the connection; or the peer might even crash. The handshake may also complete, but the peer could immediately close or cancel the connection. These possibilities need to be correctly handled in a robust application. The consequences of not doing this right include locked-up sockets (i.e., inability to accept further connections) or protocol failures.

The following code outlines the correct way to accept connections, and to recover in case of errors.

```
if (!tcp_open(&my_socket, ...))
    printf("Failed to open\n");
else while(!sock_established(&my_socket)) {
    if (!tcp_tick(&my_socket)) {
        printf("Failed to establish\n");
        break;
    }
}
if (sock_established(&my_socket)) {
    printf("Established OK!\n");
    // do whatever needs to be done...
}
```

Notice the `tcp_tick(&my_socket)` call inside the while loop. This is necessary in order to test whether the handshake was aborted by the peer, or timed out. At the end of the loop, `sock_established()` tests whether the handshake did indeed complete. If so, then the socket is ready for data flow. Otherwise, the socket should be re-opened. The same basic procedure applies for passively opened sockets (i.e., `tcp_listen()`).

3.3.4 Specifying a Listen Queue

A `tcp_socket` structure can handle only a single connection at any one time. However, a passively opened socket may be required to handle many incoming connection requests without undue delay. To help smoothly process successive connection requests with a single listening socket, you can specify that certain TCP port numbers have an associated “pending connection” queue. If there is no queue, then incoming requests will be cancelled if the socket is in use. If there is a queue, then the new connections will be queued until the current active connection is terminated.

To accept new connection requests when the passively opened socket is currently connected, use the function `tcp_reserveport()`. It takes one parameter, the port number where you want to accept connections. When a connection to that port number is requested, the 3-way handshaking is done even if there is not yet a socket available. When replying to the connection request, the window parameter in the TCP header is set to zero, meaning, “I can take no bytes of data at this time.” The other side of the connection will wait until the value in the window parameter indicates that data can be sent. Using the companion function, `tcp_clearreserve(port number)`, causes TCP/IP to treat a connection request to the port in the conventional way. The macro `USE_RESERVEDPORTS` is defined by default. It allows the use of these two functions.

When using `tcp_reserveport()`, the 2MSL (maximum segment lifetime) waiting period for closing a socket is avoided.

3.4 TCP Socket Functions

There are many functions that can be applied to an open TCP socket. They fall into three main categories: Control, Status, and I/O.

3.4.1 Control Functions for TCP Sockets

These functions change the status of the socket or its I/O buffer.

- `sock_abort`
- `sock_close`
- `sock_flush`
- `sock_flushnext`
- `tcp_extlisten`
- `tcp_extopen`
- `tcp_listen`
- `tcp_open`

The open and listen functions have been explained in previous sections.

Call `sock_close()` to end a connection. This call may not immediately close the connection because it may take some time to send the request to end the connection and receive the acknowledgements. If you want to be sure that the connection is completely closed before continuing, call `tcp_tick()` with the socket structure’s address. When `tcp_tick()` returns zero, then the socket is completely closed. Please note that if there is data left to be read on the socket, the socket will not completely close.

Call `sock_abort()` to cancel an open connection. This function will cause a TCP reset to be sent to the other end, and all future packets received on this connection will be ignored.

For performance reasons, data may not be immediately sent from a socket to its destination. If your application requires the data to be sent immediately, you can call `sock_flush()`. This function will try sending any pending data immediately. If you know ahead of time that data needs to be sent immediately,

call `sock_flushnext()` on the socket. This function will cause the next set of data written to the socket to be sent immediately, and is more efficient than `sock_flush()`.

3.4.2 Status Functions for TCP Sockets

These functions return useful information about the status of either a socket or its I/O buffers.

- `sock_alive`
- `sock_bytesready`
- `sock_dataready`
- `sock_established`
- `sock_iface`
- `sock_rbleft`
- `sock_rbsize`
- `sock_rbused`
- `sock_tbleft`
- `sock_tbsize`
- `sock_tbused`
- `tcp_tick`

`tcp_tick()` is the daemon that drives the TCP/IP stack, but it also returns status information. When you supply `tcp_tick()` with a pointer to a `tcp_Socket` (a structure that identifies a particular socket), it will first process packets and then check the indicated socket for an established connection. `tcp_tick()` returns zero when the socket is completely closed. You can use this return value after calling `sock_close()` to determine if the socket is completely closed.

```
sock_close(&my_socket);
while(tcp_tick(&my_socket)) {
    // you can do other things here while waiting for the socket to be completely closed
}
```

The status functions can be used to avoid blocking when using `sock_write()` and some of the other I/O functions. As illustrated in the following code, you can make sure that there is enough room in the buffer before adding data with a blocking function.

```
if(sock_tbleft(&my_socket,size)) {
    sock_write(&my_socket,buffer,size);
}
```

The following block of code ensures that there is a string terminated with a new line in the buffer, or that the buffer is full before calling `sock_gets()`:

```
sock_mode(&my_socket,TCP_MODE_ASCII);
if(sock_bytesready(&my_socket) != -1) {
    sock_gets(buffer,MAX_BUFFER);
}
```

3.4.3 I/O Functions for TCP Sockets

These functions handle all I/O for a TCP socket.

- `sock_aread`
- `sock_axread`
- `sock_fastread`
- `sock_getc`
- `sock_gets`
- `sock_awrite`
- `sock_axwrite`
- `sock_fastwrite`
- `sock_putc`
- `sock_puts`
- `sock_read`
- `sock_write`
- `sock_xfastread`
- `sock_xfastwrite`
- `sock_preread`

There are two modes of reading and writing to TCP sockets: ASCII and binary. By default, a socket is opened in binary mode, but you can change the mode with a call to `sock_mode()`.

When a socket is in ASCII mode, it is assumed that the data is an ASCII stream with record boundaries on the newline characters for some of the functions. This behavior means `sock_bytesready()` will return ≥ 0 only when a complete newline-terminated string is in the buffer or the buffer is full. The `sock_puts()` function will automatically place a newline character at the end of a string, and the `sock_gets()` function will strip the newline character.

Do not use `sock_gets()` in binary mode.

3.5 UDP Socket Overview

The UDP protocol is useful when sending messages where either a lost message does not cause a system failure or is handled by the application. Since UDP is a simple protocol and you have control over the retransmissions, you can decide if you can trade low latency for high reliability.

Broadcast Packets

UDP can send broadcast packets (i.e., to send a packet to a number of computers on the same network). This is accomplished by setting the remote IP address to -1, in either a call to `udp_open()` or a call to `udp_sendto()`. When used properly, broadcasts can reduce overall network traffic because information does not have to be duplicated when there are multiple destinations.

Checksums

There is an optional checksum field inside the UDP header. This field verifies the header and the data. This feature can be disabled on a reliable network where the application has the ability to detect transmission errors. Disabling the UDP checksum can increase the performance of UDP packets moving through the TCP/IP stack. This feature can be modified by:

```
sock_mode(s, UDP_MODE_CHK); // enable checksums
sock_mode(s, UDP_MODE_NOCHK); // disable checksums
```

The first parameter is a pointer to the socket's data structure, either `tcp_Socket` or `udp_Socket`.

In Dynamic C version 7.20, some convenient macros offer a safer, faster alternative to using `sock_mode()`. They are `udp_set_chk(s)` and `udp_set_nochk(s)`.

Improved Interface

With Dynamic C version 7.05 there is a redesigned UDP API. The new interface is incompatible with the previous one. "UDP Socket Functions (7.05 and later)" on page 50 covers the new interface and "UDP Socket Functions (pre 7.05)" on page 51 covers the previous one. See "Porting Programs from the older UDP API to the new UDP API" on page 52 for information on porting an older program to the new UDP interface.

3.6 UDP Socket Functions (7.05 and later)

Starting with Dynamic C 7.05, the UDP implementation is a true record service. It receives distinct datagrams and passes them as such to the user program. The socket I/O functions available for TCP sockets will no longer work for UDP sockets.

3.6.1 Control Functions for UDP Sockets

These functions change the status of the socket or its I/O buffer.

- `udp_close`
- `udp_extopen`
- `udp_open`

3.6.2 Status Function for UDP Sockets

These functions return useful information about the status of either a socket or its I/O buffers.

- `sock_bytesready`
- `sock_dataready`
- `sock_rbleft`
- `sock_rbsize`
- `sock_rbused`
- `udp_peek`

For a UDP socket, `sock_bytesready()` returns the number of bytes in the next datagram in the socket buffer, or -1 if no datagrams are waiting. Note that a return of 0 is valid, since a datagram can have 0 bytes of data.

3.6.3 I/O Functions for UDP Sockets

These functions handle datagram-at-a-time I/O:

- `udp_recv`
- `udp_recvfrom`
- `udp_send`
- `udp_sendto`

The write function, `udp_sendto()`, allows the remote IP address and port number to be specified. The read function, `udp_recvfrom()`, identifies the IP address and port number of the host that sent the datagram. There is no longer a UDP read function that blocks until data is ready.

3.7 UDP Socket Functions (pre 7.05)

This interface is basically the TCP socket interface with some additional functions for simulating a record service. Some of the TCP socket functions work differently for UDP because of its connectionless state. The descriptions for the applicable functions detail these differences.

3.7.1 I/O Functions for UDP Sockets

Prior to Dynamic C 7.05, the functions that handle UDP socket I/O are mostly the same functions that handle TCP socket I/O.

- `sock_fastread`
- `sock_fastwrite`
- `sock_getc`
- `sock_gets`
- `sock_preread`
- `sock_putc`
- `sock_puts`
- `sock_read`
- `sock_recv`
- `sock_recv_from`
- `sock_recv_init`
- `sock_write`
- `udp_close`
- `udp_open`

Notice that there are three additional I/O functions that are only available for use with UDP sockets: `sock_recv()`, `sock_recv_from()` and `sock_recv_init()`. The status and control functions that are available for TCP sockets also work for UDP sockets, with the exception of the open functions, `tcp_listen()` and `tcp_open()`.

3.7.2 Opening and Closing a UDP Socket

`udp_open()` takes a remote IP address and a remote port number. If they are set to a specific value, all incoming and outgoing packets are filtered on that value (i.e., you talk only to the one remote address).

If the remote IP address is set to -1, the UDP socket receives packets from any valid remote address, and outgoing packets are broadcast. If the remote IP address is set to 0, no outgoing packets may be sent until a packet has been received. This first packet completes the socket, filling in the remote IP address and port number with the return address of the incoming packet. Multiple sockets can be opened on the same local port, with the remote address set to 0, to accept multiple incoming connections from separate remote hosts. When you are done communicating on a socket that was started with a 0 IP address, you can close it with `sock_close()` and reopen to make it ready for another source.

3.7.3 Writing to a UDP Socket

Prior to Dynamic C 7.05, the normal socket functions used for writing to a TCP socket will work for a UDP socket, but since UDP is a significantly different service, the result could be different. Each atomic write—`sock_putc()`, `sock_puts()`, `sock_write()`, or `sock_fastwrite()`—places its data into a single UDP packet. Since UDP does not guarantee delivery or ordering of packets, the data received may be different either in order or content than the data sent. Packets may also be duplicated if they cross any gateways. A duplicate packet may be received well after the original.

3.7.4 Reading From a UDP Socket

There are two ways to read UDP packets prior to Dynamic C 7.05. The first method uses the same read functions that are used for TCP: `sock_getc()`, `sock_gets()`, `sock_read()`, and `sock_fastread()`. These functions will read the data as it came into the socket, which is not necessarily the data that was written to the socket.

The second mode of operation for reading uses the `sock_recv_init()`, `sock_recv()`, and `sock_recv_from()` functions. The `sock_recv_init()` function installs a large buffer area that gets divided into smaller buffers. Whenever a datagram arrives, it is stuffed into one of these new buffers. The `sock_recv()` and `sock_recv_from()` functions scan these buffers. After calling `sock_recv_init` on the socket, you should not use `sock_getc()`, `sock_read()`, or `sock_fastread()`.

The `sock_recv()` function scans the buffers for any datagrams received by that socket. If there is a datagram, the length is returned and the user buffer is filled, otherwise `sock_recv()` returns zero.

The `sock_recv_from()` function works like `sock_recv()`, but it allows you to record the IP address where the datagram originated. If you want to reply, you can open a new UDP socket with the IP address modified by `sock_recv_from()`.

3.7.5 Porting Programs from the older UDP API to the new UDP API

To update applications written with the older-style UDP API, use the mapping information in the following table.

UDP API prior to Dynamic C 7.05	UDP API starting with Dynamic C 7.05
MAX_SOCKETS	MAX_UDP_SOCKET_BUFFERS and MAX_TCP_SOCKET_BUFFERS
SOCK_BUF_SIZE	UDP_BUF_SIZE and TCP_BUF_SIZE
udp_open()	udp_open()
sock_write(), sock_fastwrite()	udp_send() or udp_sendto()
sock_read() (blocking function)	udp_recv() or udp_recvfrom() (nonblocking functions)
sock_fastread()	udp_recv() or udp_recvfrom()
sock_recv_init()	udp_extopen() (to specify your own buffer)
sock_recv()	udp_recv()
sock_recv_from()	udp_recvfrom()
sock_close()	sock_close() or udp_close()
sock_bytesready()	sock_bytesready()
sock_dataready()	sock_dataready()

3.8 Skeleton Program

The following program is a general outline for a Dynamic C TCP/IP program. The first couple of defines set up the default IP configuration information. The “memmap” line causes the program to compile as much code as it can in the extended code window. The “use” line causes the compiler to compile in the Dynamic C TCP/IP code using the configuration data provided above it.

Program Name: Samples\tcpip\icmp\pingme.c

```
/*
 * Starting with Dynamic C 7.30, the network addresses are initialized by defining the
 * following macro to identify the desired configuration in the file tcp_config.lib.
 */
#define TCPCONFIG 1 // static configuration of single Ethernet interface.
/*
 * Prior to Dynamic C 7.30, you must change the following values to whatever
 * your local IP address, netmask, and gateway are. Contact your network
 * administrator for these numbers.
 */
// #define MY_IP_ADDRESS "10.10.6.101"
// #define MY_NETMASK "255.255.255.0"
// #define MY_GATEWAY "10.10.6.19"
#memmap xmem
#use dcrtcp.lib
main()
{
    sock_init();
    for (;;) {
        tcp_tick(NULL);
    }
}
```

To run this program, start Dynamic C and open the Samples\TCPIP\ICMP\PINGME.C file. If you are using a Dynamic C version prior to 7.30, edit the MY_IP_ADDRESS, MY_NETMASK, and MY_GATEWAY macros to reflect the appropriate values for your device. Otherwise, edit your tcpconfig.lib (or custom_config.lib) file with appropriate network addresses for your device and define TCPCONFIG to access the desired configuration information.

Run the program and try to run ping 10.10.6.101 from a command line on a computer on the same physical network, replacing 10.10.6.101 with your value for MY_IP_ADDRESS.

3.8.1 TCP/IP Stack Initialization

The main() function first initializes the TCP/IP stack with a call to sock_init(). This call initializes internal data structures and enables the Ethernet chip, which will take a couple of seconds with the RealTek chip. At this point, the TCP/IP stack is ready to handle incoming packets.

3.8.2 Packet Processing

Incoming packets are processed whenever `tcp_tick()` is called. The user-callable functions that call `tcp_tick()` are: `tcp_open`, `udp_open`, `sock_read`, `sock_write`, `sock_close`, and `sock_abort`. Some of the higher-level protocols, e.g., `HTTP.LIB` will call `tcp_tick()` automatically.

Call `tcp_tick()` periodically in your program to ensure that the TCP/IP stack has had a chance to process packets. A rule of thumb is to call `tcp_tick()` around 10 times per second, although slower or faster call rates should also work. The Ethernet interface chip has a large buffer memory, and TCP/IP is adaptive to the data rates that both ends of the connection can handle; thus the system will generally keep working over a wide variety of tick rates.

3.9 TCP/IP Daemon: `tcp_tick()`

`tcp_tick()` is a fundamental function for the TCP/IP library. It has two uses: it drives the “background” processing necessary to maintain up-to-date information; and it may also be used to test TCP socket state. The latter use is described in the next section.

Note that `tcp_tick()` does more than just TCP processing: it is also necessary for UDP and other internal protocols such as ARP and ICMP. It also (as of Dynamic C 7.30) controls interface status.

The computing time consumed by each call to `tcp_tick()` varies. Rough numbers are less than a millisecond if there is nothing to do, tens of milliseconds for typical packet processing, and hundreds of milliseconds under exceptional circumstances. In general, the more active sockets that are in use simultaneously, the longer it will take for `tcp_tick()` to complete, however there is not much increase for reasonable numbers of sockets.

It is recommended that you call `tcp_tick()` at the head of the main application processing loop. If you have any other busy-wait loops in your application, you should arrange for `tcp_tick()` to be called in each such loop. TCP/IP library functions that are documented as “blocking” will always include calls to `tcp_tick()`, so you do not have to worry about it. Library functions which are documented as “non-blocking” (e.g., `sock_fastread()`) do not in general call `tcp_tick()`, so your application will need to do it.

Some of the provided application protocols (such as HTTP and FTP) have their own “tick” functions (e.g., `http_handler()` and `ftp_tick()`). When you call such a function, there is no need to call `tcp_tick()` since the other tick function will always do this for you.

3.9.1 `tcp_tick()` for Robust Applications

It goes without saying that your application should be designed to be robust. You should be aware that an open TCP socket may become disconnected at any time. The disconnection can arise because of a time-out (caused by network problems), or because the peer application sent a RST (reset) flag to abort the connection, the interface went down, or even because another part of your application called `sock_abort()`. Your application should check for this condition, preferably in the main socket processing loop, by calling `tcp_tick()` with the socket address. Since `tcp_tick()` needs to be called regularly, this does not add much overhead if you have a single socket. For applications which manage multiple sockets, you can use the `sock_alive()` function (new for Dynamic C 7.30). If `tcp_tick()` or `sock_alive()` returns zero for a socket, then the socket may be re-opened after your application recovers.

Regular checking of socket status is also convenient in that it can simplify the rest of your application. In effect, checking socket status in your main application loop concentrates socket error handling at a single point in the code. There is less need to perform error handling after other calls to TCP/IP functions. For example, the `sock_fastread()` function normally returns a non-negative value, but it can return -1 if there is a problem with the socket. An application function which calls `sock_fastread()` needs to check for this code, however it can choose to merely return to the caller (the main loop) if this code is detected, rather than handling the error at the point where it was first detected. This works because if `sock_fastread()` returns -1, `tcp_tick()` will return zero for that socket.

3.9.2 Global Timer Variables

The TCP/IP stack depends on the values for `MS_TIMER`, and `SEC_TIMER`. Problems may be encountered if the application program changes these values during execution.

3.10 State-Based Program Design

An efficient design strategy is to create a state machine within a function and pass the socket's data structure as a function parameter. This method allows you to handle multiple sockets without the services of a multitasking kernel. This is the way the `HTTP.LIB` functions are organized. Many of the common Internet protocols fit well into this state machine model.

The general states are:

- Waiting to be initialized.
- Waiting for a connection.
- Connected states that perform the real work.
- Waiting for the socket to be closed.

An example of state-based programming is `SAMPLES\TCPIP\STATE.C`. This program is a basic Web server that should work with most browsers. It allows a single connection at a time, but can be extended to allow multiple connections.

In general, when defining the set of states for a socket connection, you will need to define a state for each point where the application needs to wait for some external event. At a minimum, this will include states when waiting for:

- session establishment
- new received data
- space in the transmit buffer for write data
- session termination

For non-trivial application protocols, the states in-between session establishment and session termination may need to be embellished into a set of sub-states which reflect the stage of processing of input or output. Sometimes, input and output states may need to overlap. If they do not, then you typically have a step-by-step protocol. Otherwise, you have an application that uses receive and transmit independently. Step-by-step protocols are easier to implement, since there is no need to be able to overlap two (or more) sets of state.

For read states, which are waiting for some data to come in from the peer, you will typically call one of the non-blocking socket read functions to see if there is any data available. If you are expecting a fixed length of data (e.g., a C structure encoded in the TCP data stream), then it is most convenient to use the `sock_aread()` function which was introduced in Dynamic C 7.30. Otherwise, if you cannot tell how much data will be required to go to the next state, then you will have to call `sock_preread()` to check the current data, without prematurely extracting it from the socket receive buffer.

For write states, you can just keep calling `sock_fastwrite()` until all the data for this state is written. If you have a fixed amount of data, `sock_awrite()` is more convenient since you do not have to keep track of partially written data.

3.10.1 Blocking vs. Non-Blocking

There is a choice between blocking and non-blocking functions when doing socket I/O.

3.10.1.1 Non-Blocking Functions

The `sock_fastread()` and `sock_preread()` functions read all available data in the buffers, and return immediately. Similarly, the `sock_fastwrite()` function fills the buffers and returns the number of characters that were written. When using these functions, you must ensure that all of the data were written completely.

```
offset=0;
while(offset<len) {
    bytes_written = sock_fastwrite(&s, buf+offset, len-offset);
    if(bytes_written < 0) {
        // error handling
    }
    offset += bytes_written;
}
```

3.10.1.2 Blocking Functions

The other functions (`sock_getc()`, `sock_gets()`, `sock_putc()`, `sock_puts()`, `sock_read()` and `sock_write()`) do not return until they have completed or there is an error. If it is important to avoid blocking, you can check the conditions of an operation to ensure that it will not block.

```
sock_mode(socket, TCP_MODE_ASCII);
// ...
if (sock_bytesready(&my_socket) != -1){
    sock_gets(buffer, MAX_BUFFER);
}
```

In this case `sock_gets()` will not block because it will be called only when there is a complete new line terminated record to read.

3.11 TCP and UDP Data Handlers

Starting with Dynamic C 7.30¹, your application can specify data handler callback functions for TCP and UDP sockets. The data handler callback may be specified as a parameter to the `tcp_open()`, `tcp_extopen()`, `tcp_listen()`, `tcp_extlisten()`, `udp_open()`, `udp_extopen()` and `udp_waitopen()` functions.

The UDP data handler callback is always available. The TCP handler is only available if you `#define TCP_DATAHANDLER` before including `dcrtcp.lib`. Both types of callback use the same function prototype, however, the parameters are interpreted slightly differently.

The prototype for a suitable callback function is:

```
int my_data_handler(  
    int event,  
    void * socket,  
    ll_Gather * g,  
    void * info  
);
```

“event” indicates the type of callback. It is one of a predefined set of constants specified in the table below.

“socket” is a pointer to the socket structure (TCP or UDP). “g” contains a number of fields which may be accessed to find additional information, including the data stream or packet. “info” points to a structure which depends on the type of socket: `_udp_datagram_info` if the socket is UDP, or `NULL` for TCP sockets.

1. Data handler pointers were provided to the `tcp_open` etc. functions prior to this release, however the interface was not documented, and does not work in the way described herein.

The `ll_Gather` structure is defined and documented in `NET.LIB`. It is printed here for reference:

```
typedef struct {
    byte iface;           // Destination interface
    byte spare;
    word len1;           // Length of root data section
    void * data1;        // Root data (e.g., link, IP, transport headers)
    word len2;           // Length of first xmem section
    long data2;          // First xmem data extent (physical address)
    word len3;           // Length of second xmem section
    long data3;          // Second xmem data extent (physical address)
} ll_Gather;
```

The `_udp_datagram_info` is defined in `UDP.LIB`. It is documented with the `udp_peek()` function.

For UDP sockets, the callback is invoked for each packet received by the socket. For TCP sockets, the callback is invoked whenever *new* data is available that could otherwise be returned by `sock_fastread()`.

The advantages of using the data handler callback are:

- Less application overhead calling `sock_dataready()` or `sock_fastread()`
- Data copy to root buffers can be avoided
- Ability to transform data in the socket buffer (e.g., decryption)
- For UDP, may avoid the need to copy incoming data into the socket receive buffer
- Minimizes latency between `tcp_tick()` receive processing, and application processing
- Allows event-driven programming style

The following table lists the parameters to the callback for each event type.

Table 3.1 Parameters for Each Type of Callback

event	s	g	info	notes
UDP_DH_INDATA	udp_Socket	pkt data	UDI	Normal received data
UDP_DH_ICMPMSG	udp_Socket	pkt data	UDI	ICMP message received for this socket
TCP_DH_LISTEN	tcp_Socket	NULL	NULL	Passive open call (e.g., <code>tcp_extlisten()</code>)
TCP_DH_OPEN	tcp_Socket	NULL	NULL	Active open call (e.g., <code>tcp_extopen()</code>)
TCP_DH_ESTAB	tcp_Socket	NULL	NULL	3-way handshake complete, ready for data transfer
TCP_DH_INDATA	tcp_Socket	seg data	NULL	Incoming stream data
TCP_DH_OUTBUF	tcp_Socket	NULL	NULL	New space in transmit buffer (data acknowledged by peer)

Table 3.1 Parameters for Each Type of Callback

event	s	g	info	notes
TCP_DH_INCLOSE	tcp_Socket	NULL	NULL	No further incoming data (peer sent FIN)
TCP_DH_OUTCLOSE	tcp_Socket	NULL	NULL	No further outgoing data (application closed socket, sent FIN)
TCP_DH_CLOSED	tcp_Socket	NULL	NULL	Socket completely closed
TCP_DH_ABORT	tcp_Socket	NULL	NULL	Application called sock_abort
TCP_DH_RESET	tcp_Socket	NULL	NULL	Peer sent RST flag
TCP_DH_ICMPMSG	tcp_Socket	pkt data	NULL	ICMP message associated with this socket
Other	?	?	?	Reserved for future use. Callback should always return zero.

3.11.1 UDP Data Handler

For UDP sockets, the callback is invoked as soon as a new datagram is demultiplexed to the socket. For event type `UDP_DH_INDATA`, the `ll_Gather` struct is set up with the interface number and pointers to the data in the receive buffers (not the UDP socket receive buffer, since the data has not yet been copied there). The info structure is a pointer to `_udp_datagram_info` (UDI), which is set up with the usual `udp_peek` information such as the host IP address and port number, and whether the datagram is in fact an ICMP error message. If an ICMP message is received, the event type is set to `UDP_DH_ICMPMSG`. The callback should return 0 to continue with normal processing (i.e., add the datagram to the socket buffer), or 1 to indicate that the datagram has been processed and should not be added to the socket buffer.

The data pointers in the `ll_Gather` structure are the physical address (and length) of one or two datagram fragments in the main network receive buffers. (Currently, only one address will be provided, since datagrams are reassembled before passing to the UDP handler). There is also a root data pointer in the `ll_Gather` structure, that is set to point to the IP and UDP headers of the datagram.

3.11.2 TCP Data Handler

The TCP data handler is only available if you `#define TCP_DATAHANDLER`. It is invoked with a large number of different event types. Most of the events are for significant changes in the TCP socket state. You can use these events to perform customized handling of socket open and close. Apart from `TCP_DH_INDATA` and `TCP_DH_ICMPMSG`, the `ll_Gather` structure is not passed (`g` is set to `NULL`). Currently, the `info` parameter is always null for TCP sockets.

If your callback function does not understand a particular event type, or is not interested, it should return zero. This will allow for upward compatibility if new callback events are introduced.

For convenience in coding the callback, you can use the `user_data` field in the `tcp_Socket` structure to hold some application-specific data which is to be associated with a socket instance. There is no API for accessing this field; just use `s->user_data`. This field is only available if you have defined `TCP_DATAHANDLER`, and only for TCP sockets (not UDP).

There is no guarantee on the order in which events will arrive for a socket. The exceptions are that `TCP_DH_LISTEN` or `TCP_DH_OPEN` will always be first, and `TCP_DH_CLOSED` will always be last. There is no guarantee that the callback will be invoked with `TCP_DH_INCLOSE` or `TCP_DH_OUTCLOSE` before `TCP_DH_CLOSED`.

`TCP_DH_OUTBUF` indicates that some previously transmitted data has been acknowledged by the peer. Generally, this means that there is more space available in the transmit buffer. The callback can write data to the socket using `sock_fastwrite()` and other non-blocking write functions. The available transmit buffer space may be determined by `sock_tbleft()` function. When `TCP_DH_ESTAB` is invoked, the transmit buffer is normally completely empty, so the callback can write a reasonable amount of data to start with.

The `TCP_DH_INDATA` event callback is invoked after the incoming data has been stored in the socket buffer. It is only invoked if there is *new* data available from the peer. The `ll_Gather` structure is set up with one or two physical address pointers to the new data, and the logical pointer points to the IP header of the most recent datagram which provided the new data. Usually there will be only one physical address, however there may be two if the socket buffer happens to wrap around at that point. The callback will need to be coded to handle this possibility if it is accessing the data directly out of the `xmem` buffer.

The `TCP_DH_INDATA` callback is allowed to modify the new data in-place, if desired. This may be used to provide “transparent decryption” or similar services.

There are some restrictions which apply to callback code. Primarily, it is not allowed to invoke `tcp_tick()` directly or indirectly, since that will cause recursion into `tcp_tick()`. It will be possible to call `sock_fastwrite()` or `udp_sendto()` e.g., to generate some sort of response. Since `sock_fastwrite()` needs to buffer data, there is a possibility that there may be insufficient room in the transmit buffer for the generated response. Thus the callback will need to be carefully coded to avoid getting into a buffer deadlock situation if it generates responses. It will also need to co-ordinate with the rest of the application, since the application will otherwise have to contend with the possibility of arbitrary data being inserted in the write stream by the callback.

NOTE: The application must call `sock_fastread()` or other read functions to actually remove data from the TCP socket receive buffer unless the data handler callback is coded to call `sock_fastread()` itself. If neither the data handler nor the rest of the application actually read the received data, then the TCP connection will become “blocked” in the read direction.

3.12 Multitasking and TCP/IP

Dynamic C's TCP/IP implementation is compatible with both μ C/OS-II and with the language constructs that implement cooperative multitasking: costatements and cofunctions. Note that TCP/IP is not compatible with the slice statement.

3.12.1 μ C/OS-II

The TCP/IP stack may be used with the μ C/OS-II real-time kernel. The line:

```
#use ucos2.lib
```

must appear before the line

```
#use dcrtcp.lib
```

in the application program. Also be sure to call `OSInit()` before calling `sock_init()`.

Dynamic C version 7.05 and later requires the macro `MAX_SOCKET_LOCKS` for μ C/OS-II support. If it is not defined, it will default to `MAX_TCP_SOCKET_BUFFERS + TOTAL_UDP_SOCKET_BUFFERS` (which is `MAX_UDP_SOCKET_BUFFERS + 1` if there are DNS lookups).

Buffers `xalloc'd` for socket I/O should be accounted for in `MAX_SOCKET_LOCKS`.

The TCP/IP stack requires a μ C/OS2-II task to have a minimum stack size of 2K. Recall that the number of 2K stacks is defined by `STACK_CNT_2K`. If there are problems with sending a packet, try increasing the stack size to 4K.

3.12.2 Cooperative Multitasking

The following program demonstrates the use of multiple TCP sockets with costatements.

Program Name: `costate_tcp.c`

```
// #define MY_IP_ADDRESS "10.10.6.11"
// #define MY_NETMASK "255.255.255.0"
// #define MY_GATEWAY "10.10.6.1"
#define TCPCONFIG 1
#define PORT1 8888
#define PORT2 8889
#define SOCK_BUF_SIZE 2048
#define MAX_SOCKETS 2
#define memmap xmem
#define use "dcrtcp.lib"
tcp_Socket Socket_1;
tcp_Socket Socket_2;
#define MAX_BUFSIZE 512
char buf1[MAX_BUFSIZE], buf2[MAX_BUFSIZE];

// The function that actually does the TCP work
cofunc int basic_tcp[2](tcp_Socket *s, int port, char *buf){
    auto int length, space_avaliable;
    tcp_listen(s, port, 0, 0, NULL, 0);
    // wait for a connection
    while((-1 == sock_bytesready(s)) && (0 == sock_established(s)))
        // give other tasks time to do things while we are waiting
        yield;
    while(sock_established(s)) {
        space_avaliable = sock_tbleft(s);
        // limit transfer size to MAX_BUFSIZE, leave room for '\0'
        if(space_avaliable > (MAX_BUFSIZE-1))
            space_avaliable = (MAX_BUFSIZE-1);
        // get some data
        length = sock_fastread(s, buf, space_avaliable);
        if(length > 0) { // did we receive any data?
            buf[length] = '\0'; // print it to the Stdio window
            printf("%s",buf);
            // send it back out to the user's telnet session
            // sock_fastwrite will work-we verified the space beforehand
            sock_fastwrite(s, buf, length);
        }
        yield; // give other tasks time to run
    }
    sock_close(s);
    return 1;
}
```

Program Name: costate_tcp.c (continued)

```
main() {
    sock_init();
    while (1) {
        costate {
            // Go do the TCP/IP part, on the first socket
            wfd basic_tcp[0](&Socket_1, PORT1, buf1);
        }
        costate {
            // Go do the TCP/IP part, on the second socket
            wfd basic_tcp[1](&Socket_2, PORT2, buf2);
        }
        costate {
            // drive the tcp stack
            tcp_tick(NULL);
        }
        costate {
            // Can insert application code here!
            waitFor(DelayMs(100));
        }
    }
}
```


4. Optimizing TCP/IP Performance

Once you have a TCP/IP application coded and working, it is worthwhile to tune the application to get the best possible performance. There is usually a trade-off between performance and memory usage. If more memory is available, you can specify larger data buffers to improve overall performance. Conversely, if performance is already adequate, you can reduce buffer sizes to make room for more application functionality.

Some performance improvements can be made without large increases in memory usage. To make these improvements, you will need to understand how TCP, IP and the properties of the network work and interact. This is a complex subject, which is well covered in various texts. This section concentrates on the characteristics of the Dynamic C TCP/IP stack. Most of the discussion is centered around Dynamic C version 7.30, but many of the principles apply to earlier releases. The discussion also concentrates on TCP. UDP is also mentioned where appropriate, however UDP performance is mainly determined by the application so there are not as many tuning controls available in the Dynamic C libraries for tuning UDP performance.

The type of application has a large bearing on the performance tuning options which will be most appropriate. Here are some basic types of application which have different performance requirements:

- Bulk Loader: an application which periodically uploads large amounts of data (such as a log) to a server
- Casual Server: one which just needs to process occasional commands which come in from the network. This includes “interactive” servers such as telnet.
- Master Controller: one which sends short data bursts to a number of “slave” controllers, which must be sent and processed in a timely manner
- Web Server: a web-enabled appliance
- Protocol Translator: accepts stream of data, perhaps serial, and converts to a TCP data stream, or vice-versa

All these application types have different requirements for the basic properties of a communications channel, namely bandwidth, throughput and latency.

The bandwidth of a channel is the maximum sustained rate of end-to-end data transmission, in bytes per second. A full-duplex channel has the same bandwidth in each direction, independent of data traffic flowing in the opposite direction. In a half-duplex channel, the total bandwidth is divided between both directions. Ethernet is usually half-duplex in that an Ethernet chip cannot send and receive at the same time, however some types of Ethernet can run full-duplex.

The throughput of a channel is related to bandwidth, but is used to express the amount of useful data that can be transmitted through the channel in a fixed (specified) amount of time, using a practical transport protocol (i.e., a protocol which adds some overhead to each message). Throughput generally improves as the bandwidth rises, and as the time interval increases. Throughput is always less than bandwidth for finite

time intervals or practical protocols, since there is usually some overhead to establish the connection in the first place, as well as overhead during the transmission itself.

The latency of a channel can have several definitions. For our purposes, it is the minimum possible time delay between sending of a message, its receipt by the other end, and the reception of a reply; in other words, the round-trip-time (RTT). On electrical and radio channels, the latency is related to the physical length of the link and the speed of light. On channels which are more complex than a simple electrical connection, there may also be intermediate nodes which buffer the data being transmitted: this can add delays which are much larger than the speed of light between the end nodes.

Note that round-trip times are important for most communications protocols: not only do we want to send data, but we also want to receive an acknowledgment that the other end received the data.

Some examples of real networks may be helpful here. Note that the values given for RTT are approximations since they depend on the length of the connection, the sizes of packets sent, or intermediate nodes. Throughput is specified for an infinite time interval, assuming TCP over IP with 600 bytes of data per packet, and no data in the acknowledgment. The RTT figure assumes the same size packets.

Table 4.1 Channel Characteristics for Selected Networks

Type	Bandwidth (Byte/sec)	RTT (msec)	Throughput (Byte/sec)
Local 10Base-T Ethernet	1.25M	0.6	1M
PPP over 8N1 serial (57.6k)	5760	120	5000
PPPoE over 1.5Mbit DSL	187k	4	150k

The above table does not count any delay in the host which generates the response, nor any delay passing through the Internet. These represent minimum possible RTTs.

4.1 DBP and Sizing of TCP Buffers

An important quantity derived from the above is known as Delay-Bandwidth Product (DBP). As the name suggests, this is the product of bandwidth and RTT, and has units of bytes. It represents the maximum amount of data (and overhead) that can exist “in the network” at any point in time. This number has implications for sizing of TCP socket buffers. The DBP for local 10Base-T Ethernet is about 750 bytes. For local Ethernet connections, the DBP is about the same as the packet size of the transmitted data. For wider area networks that have significant propagation delays, the DBP can increase substantially. For example, satellite links can add several 100’s of milliseconds to the RTT. If the bandwidth is high enough, the DBP can exceed the packet size by orders of magnitude. This means that several packets may be in transit at the same time.

The DBP is important for TCP connections. This is because TCP is able to transmit a large number of packets into the network without having to wait for an acknowledgement for each one. Similarly, a TCP can receive a large number of packets without necessarily acknowledging them all. In fact, TCP only has

to acknowledge the most recent packet; the sender can assume that all earlier packets are implicitly acknowledged.

How does all this apply to sizing of TCP socket buffers? It basically means that there is little point in making the buffers (both transmit and receive) larger than the expected maximum DBP of the communications channel. For connections which are expected to traverse the Internet, you may need quite large buffers. For local Ethernet only, the buffers need not be larger than, say, two packets.

The maximum packet size is a compromise between performance and memory usage. The largest packet supported by `dcrtcp.lib` is 1500 bytes, which is dictated by the limits of Ethernet. Dynamic C's default packet size is 600 bytes. Using large packet sizes improves performance for bulk data transfer, but has little effect for interactive traffic. Performance is improved for large packet sizes mainly because there is less CPU overhead per byte. There is a roughly fixed amount of CPU time required to process each packet. This is obviously better utilized if there are a large number of bytes per packet.

When using Ethernet, the Rabbit processor is limited in its overall TCP/IP throughput by CPU power.

10Base-T Ethernet is capable of 1MB/sec for TCP sockets¹, however the Rabbit 2000 running at 21MHz will only be able to transmit at about 270kB/sec when sending 1500 byte packets. Receive rate is slightly slower at about 220kB/sec. This scales approximately linearly with respect to CPU clock speed as well as application use of the CPU. In short, current Rabbit-based boards cannot use the full bandwidth of a local Ethernet link.

The situation changes for PPP over serial. In this case, the serial port bandwidth is less than the rate at which packets can be generated or received. Also, PPP is typically used to access peers over the Internet, so there may be a much larger DBP than for a pure point-to-point link. For PPP serial links, smaller packet sizes, e.g. 256 bytes, are satisfactory for bulk data transfers without impacting interactive traffic, should that be required. Socket buffer sizes should be determined based on the expected Internet RTTs, which may be 1 second or more. For a 57.6kbps serial link, the DBP is 5000 bytes for 1 second RTT, thus the socket buffers should be about this size for receive and transmit.

TCP is adaptive to changing network conditions. For example, the RTT can vary considerably at different times of day, and communication channels can become congested. TCP is designed to cope with these conditions without exacerbating any existing problems, however socket buffer and packet sizes are usually constants for the application so they need to be selected with due consideration to the most common conditions.

1. Assuming there is no other traffic on the Ethernet, and that collisions are rare. This is rarely the case, so a 50-80% utilization of bandwidth is considered the maximum desirable Ethernet load.

4.2 TCP Performance Tuning

TCP is a well-designed protocol, and provides nearly optimum performance over a wide range of conditions. Obtaining the best possible performance requires the application to co-operate with TCP by setting the correct options if the defaults are not optimal, making the most efficient use of the socket API functions, and providing appropriate memory and CPU resources.

The available performance-related options are:

- Whether to use the Nagle algorithm
- Settings for time-out values
- Whether to define a pending connection queue (“reserved port”)
- Setting the IP Type Of Service field
- Packet, buffer and MTU sizes
- ARP cache size (for Ethernet).

Sizing of buffers was discussed in the previous section. The following sections discuss the other performance controls.

4.2.1 The Nagle Algorithm

The Nagle algorithm is an option for TCP sockets. It modifies the transmit processing for a socket, but has no effect on receive processing. The TCP/IP library allows Nagle to be applied on a per-socket basis.

Most applications should leave the Nagle algorithm enabled for each TCP socket, which is the default. This provides the best utilization of bandwidth, since it prevents many small packets from being sent where one big packet would be preferable.

The main reason to override the default, and disable the Nagle algorithm, is for applications that require the least possible delay between writing data to the socket, and its receipt by the peer application. This comes at the expense of efficiency, so you should carefully consider whether the application really requires the slight reduction in delay.

When Nagle is turned off, using the macro `tcp_set_nonagle(&socket)`, transmit processing is changed so that TCP tries to transmit a packet for each call of a socket write function such as `sock_fastwrite()`.

If Nagle is on (which is the default state or can be set using `tcp_set_nagle(&socket)`) a new packet will only be sent if there is no outstanding unacknowledged data. Thus, on a slow network where acknowledgements from the peer take a substantial amount of time to arrive, fewer packets will be sent because there is a greater chance that there is some unacknowledged data.

The difference may be illustrated by the following example: suppose that a TCP socket connection is currently established and quiescent (i.e., there is no outstanding data to be acknowledged; everything is up-to-date). The network round-trip-time is 550ms. The application writes ten single characters to the socket, at 100ms intervals each. With Nagle turned off, ten packets will be sent at approximately 100ms intervals. Each packet will contain a 40-byte header (IP and TCP) with a single byte of data. A total of 410 bytes will be sent. With Nagle on, the first character written at time zero will cause a 41-byte packet to be sent. The acknowledgment of this first packet will not arrive for another 550ms. In the meantime, the application writes an additional 5 characters at 100ms intervals. Since there is outstanding unacknowledged data (the

first character) these characters are not sent immediately. They are buffered, waiting for an acknowledgment from the peer. When the first character's acknowledgment comes in at 550ms, there is no outstanding unack'ed data; the additional 5 characters have not yet been sent so they do not count as unack'ed data. Now the TCP stack will send the 5 additional characters in a single packet at approximately $t=550\text{ms}$. While that packet is in transit, 4 more characters are written by the application. Again, these characters will be buffered since characters 2 through 6 have not been acknowledged. Only when the next acknowledgment is received will these 4 characters be sent. The total number of packets sent is 3, with 1, 5 and 4 bytes of data. This translates to 130 bytes in total.

Obviously, the total number of bytes transmitted, including overhead, is far less when Nagle is used (130 compared with 410 bytes). One can also examine how this looks from the point of view of the peer.

In the non-Nagle case, each character is received 275ms after it was transmitted (we assume that the one-way trip is half of the RTT). The last character is received at $t=1175\text{ms}$ (with the reference $t=0$ taken as the first character transmission time). The acknowledgment of the last character, which completes the transaction, is received at $t=1400\text{ms}$.

In the Nagle case, the last character is received at $t=1375$ and the final acknowledgment at $t=1650$. In this example, the peer received all 10 characters 200ms later when Nagle was used.

It can be seen that at a slight cost in increased delay, a great saving in total data transmission was made. If the above example was extended to hundreds or thousands of characters, then the additional delay would remain constant at a few hundred ms, whereas the network bandwidth would be better utilized by a factor approaching five!

In conclusion, leave Nagle on unless you absolutely must have the lowest delay between transmission and reception of data. If you turn Nagle off, ensure that your application is disciplined enough to write the largest blocks it can. For example, if you have to send an 8-byte value (as a unit), construct the full 8 bytes as a single block then write them all in a single `sock_fastwrite()` call, rather than calling `sock_fastwrite()` with two 4-byte calls or, worse, 8 single byte calls.

A useful alternative to turning Nagle off is to control packetization using calls to `sock_flush()`, `sock_noflush()` and `sock_flushnext()`. These functions allow the application fairly fine control over when TCP sends packets. Basically, `sock_noflush()` is used to set a "lock" on the socket that prevents TCP from sending packets containing new data. After `sock_noflush()`, you can call `sock_fastwrite()` or other write functions. The new data will not be sent until the socket is "unlocked" with a call to `sock_flush()`. `sock_flushnext()` unlocks the socket, but TCP does not send any data until the next write function is called.

4.2.2 Time-Out Settings

There are many time-out settings in TCP. These are necessary because the TCP socket needs to be able to take meaningful actions when things take longer than expected. For good performance, it is also sometimes necessary for the socket to delay slightly some action that it could otherwise perform immediately.

The time-out settings currently apply to all sockets; they cannot be applied selectively because they are in the form of macro constants.

In general, you can improve overall TCP performance by reducing some of the time-out settings, however there is a law of diminishing returns, and you can also start to reduce overall efficiency. What may be good settings for a local Ethernet connection may be very poor for an Internet connection. Note that if you optimize time-out settings for a particular network environment, you will need to document this so that your

end-users do not inadvertently use your application in the wrong sort of environment. For this reason, it is best to use the default settings for general-purpose applications, since the defaults work well in worst-case settings without affecting best-case performance unduly.

TCP is internally adaptive to network bandwidth and RTT, which are the main variables. Some of the time-out settings only apply to an initial “guess” of the network characteristics; TCP will converge to the correct values in a short time. Specifying a good initial guess will help TCP in the initial stages of establishing a socket connection.

4.2.2.1 Time-Out Setting Constants

The following constants can be `#defined` before including `dcrtcp.lib`. They specify various time intervals that have a bearing on connection performance.

RETRAN_STRAT_TIME

This defaults to 10 ms. It specifies the minimum time interval between testing for retransmissions of data for a particular TCP socket. This not only provides an upper bound for packet transmission rate, but also cuts down on CPU overhead. Since retransmissions are basically driven from `tcp_tick()`, the less time used in `tcp_tick()` processing the more time is left for your application. Note that the actual minimum retransmit interval is defined by `TCP_MINRTO`; this setting only affects the testing interval.

Retransmissions are only required when there is an unexpected surge in network congestion, which causes packets to be delayed well beyond the average or even dropped.

It is not recommended to reduce this setting, but you could increase it to about 100ms to cut down on `tcp_tick()` overhead without materially affecting most applications.

TCP_MINRTO

Prior to Dynamic C 9.01, this macro defaulted to 250 ms; the default was changed to 10 ms. This value specifies the actual minimum time between TCP retransmissions. Reducing this will not affect performance in a properly functioning network, and may in fact worsen efficiency. Only in a network that is dropping a high percentage of packets will this setting have any real effect. On local Ethernet connections, genuine packet drops will be practically non-existent. The most likely cause of delays is if a host CPU is tied up and unable to perform network processing. On Internet connections, setting a retransmit time shorter than the default value is just as likely to worsen the congestion which is causing packets to be dropped in the first place.

The only case where this value might be profitably reduced is the case of a point-to-point link where there is a lot of packet loss (maybe because the RS232 wiring is routed near an industrial welder). In this case, any packet loss may be assumed to be because of noise or interference, not because of router congestion. In the Internet, most packet loss is because of router congestion, in which case there is nothing to be gained by reducing `TCP_MINRTO`.

Another reason for not reducing this setting is that modern TCP/IP implementations only acknowledge every second packet received (or after a short time-out - see `TCP_LAZYUPD`). Normally, this will happen within the default value time interval, so there will be no unnecessary retransmission.

TCP_TWTIMEOUT

This defaults to 2000 ms (2 seconds). This is one area where embedded system requirements conflict somewhat with recommendations in the standards documents. The “time-wait” time-out is a waiting period that is necessary when a socket is closed. This waiting period is supposed to be twice the maximum lifetime of any packet in the network. The maximum packet lifetime is 255 seconds, so the time-wait time-out should be about 8 minutes. The purpose of the waiting time is to allow both ends of the connection to be satisfied that their respective peer has agreed to the close and acknowledged it.

This wait time only affects the closed socket i.e., the unique socket combination of IP addresses and port numbers. It means that when a socket is closed, the same socket cannot be re-opened until at least 8 minutes have passed.

This is usually no problem for systems that have large memories to hold the state of recently closed sockets. For an embedded system, which has a limited pool of sockets and limited memory for storing connection states, this wait time is inconvenient since the socket structure cannot be re-used until the time-wait period has expired.

The default time-wait period is thus set to 2 seconds in the Dynamic C TCP/IP libraries. This will work perfectly well for local Ethernet connections, where the maximum packet lifetime is of the order of milliseconds. For Internet connections, this may be a bit short, but will generally be satisfactory.

If in fact the time-wait period is too short, the worst that will happen is that one of the peers will be unsure about whether the other end got the last segment of data, and confusion may happen if old packets (from this connection) happen to arrive after the close. This latter case is unlikely to happen, but if it does then it will eventually be resolved when the socket connection process times out.

If you want your application to be more robust, you can increase this value. 8 minutes is an extremely conservative value. Most implementations shorten this to 2 minutes or 30 seconds, since packets are extremely unlikely to survive more than 15 seconds.

Note that this value is only used if you do *not* specify the `tcp_reserveport()` option for the local port of a passively opened connection. If you specify `reserveport`, then the time-wait period is set to zero.

TCP_LAZYUPD

This defaults to 5 ms, and is used for several purposes. The first use is to reschedule transmission attempts that could not be processed owing to local resource shortages. For example, if a previous packet is still being transmitted via a slow PPP interface, the current packet may need to be delayed. Similarly, the Ethernet hardware can be busy. In these cases, the TCP stack needs to try again a short time later.

The second use is to allow time for further information to come in from the network before transmitting otherwise empty packets. TCP has two main reasons for transmitting packets with no data content. The first is acknowledgement of incoming data when we have nothing to send, and the other is to update our receive window to the peer. The receive window tells the peer how much data it can transmit which we can store in our socket receive buffer. This window needs to be updated not only when we receive data, but also when the application reads data out of the receive buffer.

Rather than send these empty packets as soon as possible, it is often profitable to wait a short time. In the case of window updates, this can allow the application to write some data after the read which updated the window. The data can be sent with the window update, which improves efficiency because one packet can do the work of two. For receive data acknowledgements, the same trick can be applied i.e., piggy-backing on some additional data.

These optimizations can be taken advantage of quite often with most applications, so it is worth while specifying the lazy update time-out to be at least a few ms. Lowering the lazy update interval can slightly improve latency and throughput on high-speed (i.e., local Ethernet) connections.

4.2.3 Reserved Ports

As mentioned in the `TCP_TWTIMEOUT` description, you can specify that certain TCP port numbers have the special property of being “reserved.” If a port is reserved, it has two effects:

- A number of pending connections can be queued while a socket connection is established. The pending connections form a FIFO queue, with the longest-outstanding pending connection becoming active after the current connection is closed.
- The time-wait time-out is truncated when the current connection is closed.

Together, these increase the performance of passively-opened sockets, which are designed to implement server functions such as FTP and HTTP servers. Reserving a port has no effect on actively opened sockets (i.e., “clients”), and does not affect its performance during the life of each connection.

The functions `tcp_reserveport()` and `tcp_clearreserve()` respectively enable and disable a TCP port number from being treated in this manner.

4.2.4 Type of Service (TOS)

Type Of Service is an IP (Internet Protocol) header field that causes routers in the Internet to handle packets according to the specified service level. TOS has not been widely deployed in the past, but recently Internet routers have been able to take advantage of the TOS field.

TOS generally takes one (and only one) of a pre-specified number of values. The currently available values are:

- IPTOS_DEFAULT - the default, used when none of the following are obviously applicable.
- IPTOS_CHEAP - minimize monetary cost. Used for bulk transfers where speed or reliability are not of concern, and you are paying by the packet.
- IPTOS_RELIABLE - maximize reliability.
- IPTOS_CAPACIOUS - maximize throughput.
- IPTOS_FAST - minimize delay.
- IPTOS_SECURE - maximize security.

IP does not guarantee that the TOS setting will improve the objective performance, however, it at least guarantees that performance will not be any worse than if the default TOS was selected. In other words, it doesn't hurt to specify TOS, and it may even help!

TOS can be set on a packet-by-packet basis; however, the TCP stack only allows a TOS to be set for a socket (TCP or UDP) which is used for all packets until changed. The function `sock_set_tos()` is used to set the TOS field.

4.2.5 ARP Cache Considerations

ARP (Address Resolution Protocol) is only relevant for non-PPPoE Ethernet, not PPP interfaces. Although it works in the background, mainly to translate IP addresses into Ethernet MAC addresses, there are some considerations which apply to TCP (and UDP) performance.

There is a limited size cache of address mapping entries, known as the ARP Table. The cache is necessary in order to avoid network traffic each time a socket connection is established. It must be sized appropriately to avoid "cache misses" as much as possible.

If the controller board is to be used exclusively in "server mode," i.e., TCP sockets opened passively, then the cache does not have to be very big. If, on the other hand, the controller is going to actively establish sessions with a number of hosts, then the cache should be big enough to contain an entry for each host such that entries do not get pushed out for at least a few minutes.

The ARP Table also contains special entries for routers that are on the local Ethernet. These entries are important, since they represent entries for all hosts that are not on the local LAN segment subnet.

The default sizing rule for the ARP Table allocates an entry for each interface (including point-to-point) plus 5 entries for each Ethernet interface in use. The single entry for each interface is basically reserved for routers, on the assumption that each interface will probably require a router to allow connections to hosts which are farther afield. The additional 5 entries (for Ethernet) are for non-router hosts that the controller board will need to talk to.

This implies that 5 connections to hosts on the Ethernet subnet can be supported simultaneously, without any of the entries being pushed out. If the table is full, connection to a 6th host can be made, with the least-recently-used host entry being pushed out to make room.

If your application connects with, say, ten hosts in random order, it is likely that the ARP Table will need to be increased in size. If in doubt, increase the table size, since each entry only takes up about 32 bytes.

4.3 Writing a Fast UDP Request/Response Server

UDP is a lightweight protocol wrapper that adds port number “multiplexing” and checksums to basic IP packets. Being lightweight, it is capable of being very fast, with low CPU overhead. UDP is often selected for custom application protocols that do not need the reliable, stream-oriented, connections of TCP.

UDP is connectionless, however, application designers can think in terms of client-server or transaction-based programming. A popular design for UDP servers is to have the controller board listen for incoming datagrams. Each incoming message is processed and an immediate reply is sent. It is left up to the client to retransmit messages if it did not receive a reply in the expected time frame. The server, however, is extremely simple to implement, which allows it to serve more clients than a TCP-based server could manage.

Starting with Dynamic C 7.30, a data handler facility has been added to UDP (as well as TCP) sockets. The data handler is especially efficient for UDP, since it allows the datagram to be processed without any copying to the socket buffer.

The UDP data handler is a callback function whose address is supplied on the `udp_extopen()` call. For simple request/response applications, the only application requirements are to define the data handler, and call `tcp_tick()` repeatedly in a loop after setting up the TCP/IP stack and opening the UDP socket.

The sample program `Samples\tcpip\udp\udp_echo_dh.c` shows how to implement a simple UDP echo server using the technique described in this section.

4.4 Tips and Tricks for TCP Applications

This section contains miscellaneous suggestions for getting the most out of your TCP-based applications.

Application design requirements that affect TCP performance include:

- the responsiveness and throughput requirements of the application
- how often `tcp_tick()` can be called
- whether socket is used in ASCII or binary mode
- whether multitasking or “big loop” programming style.

The list of [application types](#) at the beginning of this chapter is used as a basis for discussion. Your application may neatly fit into one of these categories, or it may be a combination of several. In either case, you should try to follow the programming guidelines unless you are fairly experienced with the Dynamic C TCP/IP libraries.

4.4.1 Bulk Loader Applications

This type of application is idle (from the TCP/IP point of view) most of the time, but this is punctuated by periods of intensive data transfer. Applications which exhibit this characteristic include data loggers and file transfer agents e.g. FTP server or client. Sending email via SMTP also comes under this category.

The main application requirement is good utilization of the available bandwidth i.e., highest throughput. This is achieved by using the largest practical buffer sizes, processing data in the largest possible chunks, and minimizing data copying. Since the Rabbit processor is CPU-bound when dealing with high speed transfers (over Ethernet), every time the data is “handled” it reduces the ultimate throughput.

The Nagle algorithm should be left ON. Time-outs should be set to generously high values to avoid unnecessary retransmissions. The TOS should be set to IPTOS_CAPACIOUS.

Bulk TCP transfers are most efficient when the packet size is the largest possible. The largest packet size is limited to the MTU size of the network connection. You can assume that 600 bytes is a reasonable MTU for Internet connections. You can use up to 1500 for all supported interface types (except PPPoE, which is limited to 1492), however it is best to use 600 if Internet connections are expected. If the Internet MTU is in fact less than the expected value, then packets may become fragmented, which lowers efficiency. You cannot do much about this except reduce the MTU.

When the MTU is determined, the maximum TCP packet data length will usually be the MTU minus 40. The 40 bytes are for the IP and TCP header overhead. For a 600 byte MTU, the maximum TCP data segment size will be 560. Thus, TCP performance will be best if data is handled in multiples of 560 bytes.

It is not quite this simple, however. When a TCP connection is opened, both sides can agree to use different data segment sizes than the default. Generally, whichever side has the smallest MTU will place a limit on the segment size. This is negotiated via the TCP MSS (Maximum Segment Size) option.

In your program, rather than hard-coding the optimum chunk size, you can define a symbol as follows:

```
#define TCP_CHUNK_SIZE (MAX_MTU - 40)
```

Where `MAX_MTU` is a symbol defined by the library to be the actual MTU in effect. For multiple interfaces, it is probably better to use the minimum value of any interface. You can find out the current MTU for an interface using `ifconfig(iface, IFG_MTU, &mtu, IFS_END)` which will read the MTU for interface “iface” into the integer variable “mtu”.

Most of the time, the TCP socket MSS will be equal to the fixed value above. In cases where it is smaller, there will not be a noticeable decrease in efficiency.

Once you determine the appropriate chunk size, use `sock_awrite()` or `sock_axwrite()` (for extended memory data) with the specified chunk size, except possibly for the last chunk. The function `sock_awrite()` and friends are available starting with Dynamic C 7.30. They have the advantage that the data is completely buffered, or not at all. `sock_fastwrite()` may buffer less than the requested amount, which means that your application needs to keep track of the current position in the data being sent. `sock_awrite()` does not do things “by halves,” so it is easier to keep track in the application. Because it will not do small data moves, it is also slightly more efficient in terms of CPU time.

4.4.2 Casual Server Applications

A casual server is a term we use for applications that need to respond to occasional requests for information, or commands, without large data transfers. Although the amount of data transfer is limited, the application still needs to be as responsive as possible. Example applications of this type include machine, building and power controllers. Interactive servers are also included, such as telnet.

The main goal here is to achieve low latency.

4.4.3 Master Controller Applications

Master controllers are responsible for coordinating access to a number of other devices (via TCP/IP or other types of communication) or acting as an “access concentrator”. Data transfer may be low to moderate. Latency should be minimized.

4.4.4 Web Server Applications

The TCP/IP libraries include web server software. HTTP . LIB takes advantage of the TCP library to get good performance. Your application can still affect web server performance, since it may be responsible for generating content via CGI callback functions. Web servers have much the same characteristics as “bulk loaders,” however, they are such a common case that they deserve special treatment.

4.4.5 Protocol Translator Applications

A protocol translator basically converts between a TCP data stream and some other type of data stream, for example asynchronous serial data. The data may flow in either or both directions.

This type of application has the most stringent requirements on both throughput and latency. This is because the incoming stream may not be amenable to any sort of flow control: it is necessary for TCP to keep up with a possibly high data rate. Also, the more timely the transmission of data, the more useful the protocol translator.

5. Network Addressing: ARP & DNS

ARP (Address Resolution Protocol) and DNS (Domain Name System) perform translations between various network address formats. ARP converts between IP addresses and (usually) Ethernet hardware addresses. DNS converts between human-readable domain names such as “ftp.mydomain.org” and IP addresses.

ARP and DNS are not closely related protocols, but they are lumped together in this chapter for convenience. In the Dynamic C TCP/IP libraries, `ARP.LIB` handles ARP proper, as well as router (gateway) functionality.

5.1 ARP Functions

ARP (Address Resolution Protocol) is used on non-PPPoE Ethernet interfaces. ARP is used to determine the hardware address of network interface adapters. Most of the ARP functionality operates in the background and is handled by the TCP/IP libraries. Most applications should not need to deal with ARP, and indeed some of the ARP functions are quite complex to use correctly.

Nevertheless, there are some useful debugging functions included in `ARP.LIB`.

Starting with Dynamic C 7.20, the internal ARP processing was converted to non-blocking style. This has no direct impact on applications, except that there will be lower maximum latency in `tcp_tick()` calls.

The ARP functions are all named starting with `_arp`, `arpcache`, `arpresolve`, or `router`.

`router_printall()` is a useful function for debugging router table problems, for example in the case where connections to hosts which are not on local subnets appear to be failing.

5.2 Configuration Macros for ARP

ARP_LONG_EXPIRY

Number of seconds that a normal entry stays current. Defaults to 1200.

ARP_SHORT_EXPIRY

Number of seconds that a volatile entry stays current. Defaults to 300.

ARP_PURGE_TIME

Number of seconds until a flushed entry is actually deleted. Defaults to 7200.

ARP_PERSISTENCE

Number of retries allowed for an active ARP resolve request to come to fruition. Default is 4. If no response is received after this many requests, then the host is assumed to be dead. Set to a number between 0 and 7. This number relates to the total time spent waiting for a response as follows:

$$\text{timeout} = 2^{(\text{ARP_PERSISTENCE}+1)} - 1$$

For example, for 0 the time-out is 1 second. For 4 it is 31 seconds. For 7 it is 255 seconds. If you set this to 8 or higher, then ARP will persist forever, retrying at 128 second intervals.

ARP_NO_ANNOUNCE

Configuration items not defined by default. Do not announce our hardware address at `sock_init()`.

This macro is undefined by default. Do not uncomment it in `NET.LIB`. Instead, define it in your mainline C program before including the networking libraries.

ARP_CONFLICT_CALLBACK

Define a function to call in case of IP address conflict. This function takes a `arp_Header` pointer as the first and only parameter. It should return one of

- 0: do not take any action
- `0xFFFFFFFF` : abort all open sockets with `NETERR_IPADDR_CONFLICT`
- other: new IP address to use. Open sockets aborted with `NETERR_IPADDR_CHANGE`.

This macro is undefined by default. Do not uncomment it in `NET.LIB`. Instead, define it in your mainline C program before including the networking libraries.

ARP_TABLE_SIZE

Define to the number of ARP table entries. The default is set to the number of interfaces, plus 5 entries for every non-PPPoE Ethernet interface. The maximum allowable value is 200.

ARP_ROUTER_TABLE_SIZE

Define the maximum number of routers. Defaults to the number of interfaces, plus an extra entry for each non-PPPoE Ethernet.

5.3 DNS Functions

Starting with Dynamic C 7.05, non-blocking DNS lookups are supported. Prior to DC 7.05, there was only the blocking function, `resolve()`. Compatibility has been preserved for `resolve()`, `MAX_DOMAIN_LENGTH`, and `DISABLE_DNS`.

The application program has to do two things to resolve a host name:

1. Call `resolve_name_start()` to start the process.
2. Call `resolve_name_check()` to check for a response.

Call `resolve_cancel()` to cancel a pending lookup.

5.4 Configuration Macros for DNS Lookups

DISABLE_DNS

If this macro is defined, DNS lookups will not be done. The DNS subsystem will not be compiled in, saving some code space and memory.

DNS_MAX_RESOLVES

4 by default. This is the maximum number of concurrent DNS queries. It specifies the size of an internal table that is allocated in xmem.

DNS_MAX_NAME

64 by default. Specifies the maximum size in bytes of a host name that can be resolved. This number includes any appended default domain and the NULL-terminator. Backwards compatibility exists for the `MAX_DOMAIN_LENGTH` macro. Its value will be overridden with the value `DNS_MAX_NAME` if it is defined.

For temporary storage, a variable of this size must be placed on the stack in DNS processing. Normally, this is not a problem. However, for μ C/OS-II with a small stack and a large value for `DNS_MAX_NAME`, this could be an issue.

DNS_MAX_DATAGRAM_SIZE

512 by default. Specifies the maximum length in bytes of a DNS datagram that can be sent or received. A root data buffer of this size is allocated for DNS support.

DNS_RETRY_TIMEOUT

2000 by default. Specifies the number of milliseconds to wait before retrying a DNS request. If a request to a nameserver times out, then the next nameserver is tried. If that times out, then the next one is tried, in order, until it wraps around to the first nameserver again (or runs out of retries).

DNS_NUMBER_RETRIES

2 by default. Specifies the number of times a request will be retried after an error or a time-out. The first attempt does not constitute a retry. A retry only occurs when a request has timed out, or when a nameserver returns an unintelligible response. That is, if a host name is looked up and the nameserver reports that it does not exist and then the DNS resolver tries the same host name with or without the default domain, that does not constitute a retry.

DNS_MIN_KEEP_COMPLETED

10000 by default. Specifies the number of milliseconds a completed request is guaranteed to be valid for `resolve_name_check()`. After this time, the entry in the internal table corresponding to this request can be reused for a subsequent request.

DNS SOCK BUF SIZE

1024 by default. Specifies the size in bytes of an xmem buffer for the DNS socket. Note that this means that the DNS socket does not use a buffer from the socket buffer pool.

6. IGMP and Multicasting

The Internet Group Management Protocol (IGMP) and multicasting are supported by the Dynamic C TCP/IP stack starting with version 7.30.

6.1 Multicasting

Multicasting is a form of limited broadcast. UDP is used to send datagrams to all hosts that belong to what is called a “host group.” A host group is a set of zero or more hosts identified by the same destination IP address. The following statements apply to host groups.

- Anyone can join or leave a host group at will.
- There are no restrictions on a host’s location.
- There are no restrictions on the number of members that may belong to a host group.
- A host may belong to multiple host groups.
- Non-group members may send UDP datagrams to the host group.

Multicasting is useful when data needs to be sent to more than one other device. For instance, if one device is responsible for acquiring data that many other devices need, then multicasting is a natural fit. Note that using multicasting as opposed to sending the same data to individual devices uses less network bandwidth.

6.1.1 Multicast Addresses

A multicast address is a class D IP address, i.e., the high-order four bits are “1110.” Addresses range from 224.0.0.0 to 239.255.255.255. The address 224.0.0.0 is guaranteed not to be assigned to any group, and 224.0.0.1 is assigned to the permanent group of all IP hosts (including gateways). This is used to address all multicast hosts on a directly connected network.

6.1.2 Host Group Membership

Any datagram sent to a multicast address is received by all hosts that have joined the multicast group associated with that address. A host group is joined automatically when the remote IP address passed to `udp_open()` is a valid multicast address. A host group may also be joined by a call to `multicast_joingroup()`. Leaving a host group is done automatically when `udp_close()` is called. Like joining, leaving a group may be done explicitly by an application by calling an API function, in this case: `multicast_leavegroup()`.

6.2 IGMP

As long as all multicast traffic is local (i.e., on the same LAN) IGMP is not needed. IGMP is used for reporting host group memberships to any routers in the neighborhood. The library `IGMP.LIB` conforms to RFC 2236 for IGMPv2 hosts.

6.3 Multicast Macros

As mentioned above, the use of IGMP is not required for multicast support on a LAN. You may select only multicast support by defining `USE_MULTICAST`.

USE_MULTICAST

This macro will enable multicast support. In particular, the extra checks necessary for accepting multicast datagrams will be enabled and joining and leaving multicast groups (and informing the Ethernet hardware about it) will be added.

USE_IGMP

If this macro is defined, the `USE_MULTICAST` macro is automatically defined. This macro enables sending reports on joining multicast addresses and responding to IGMP queries by multicast routers. Unlike `USE_MULTICAST`, this macro must be defined to be 1 or 2. This indicates which version of IGMP will be supported. Note, however, that both version 1 and 2 IGMP clients will work with both version 1 and 2 IGMP routers. Most users should just choose version 2.

IGMP_V1_ROUTER_PRESENT_TIMEOUT

Defaults to 400. When IGMPv2 is supported, a timer is set to this many seconds every time the board sees an IGMPv1 message from an IGMP router. As long as there is time left on the timer, the board acts as an IGMPv1 host. If the timer expires, the board returns to acting as an IGMPv2 host.

IGMP_UNSOLICITED_REPORT_INTERVAL

Defaults to 100 deciseconds (10 seconds). This value is specified in deciseconds. It determines the maximum random interval between the initial join report for a multicast group and the second join report.

7. PPP Driver

The PPP packet driver is a set of libraries in Dynamic C that allows the user to establish a PPP (Point-to-Point Protocol) link over a full-duplex serial line between a Rabbit-based controller and another system that supports PPP. You may also establish PPP links over Ethernet (PPPoE).

A common use of the PPP protocol is the transfer of IP packets between a remote host and an Internet Service Provider (ISP) over a modem connection. The PPP packet driver supports the transfer of Internet Protocol (IP) data and is compatible with all TCP/IP libraries for the Rabbit.

Establishing PPP links has become easier and more flexible. You can have as many different PPP interfaces as you have available serial (and Ethernet) ports. You can also run PPP (over serial and/or Ethernet) at the same time as ordinary non-PPPoE Ethernet.

7.1 PPP Libraries

The PPP driver is in three library files, though it is mainly controlled via the `ifconfig()` function and friends. PPP is mainly controlled via the `ifconfig()` function at runtime and `IFCONFIG_PPP*` configuration macros at compile time.

PPP.LIB

Contains routines to handle the link negotiation (LCP), authentication (PAP) and IP negotiation (IPCP). These are the three main sub-protocols of PPP. `PPP.LIB` calls routines in the other two libraries to handle the lower level (physical) layer.

PPPLINK.LIB

Contains handlers for the asynchronous serial physical layer, namely the interrupt service routine for transmitting and receiving characters over the serial link. It also handles the insertion and detection of escape characters and CRC generation and checking.

PPPOE.LIB

Contains handlers for the PPPoE physical layer, which is mainly the access concentrator discovery mechanism, and the addition of the PPPoE header to Ethernet packets. This library calls the Ethernet driver library to handle the Ethernet physical layer.

7.2 External Modem Library

A fourth library, `MODEM.LIB`, contains functions for controlling an external modem through a full RS232 link. `MODEM.LIB` should not be required for most ISP connections, since most ISPs these days auto-detect the use of PPP and do not require any special logon screen navigation. Basic dial-up to an ISP is handled directly by `ifconfig()` settings, which do not require any special modem control providing that your modem has a Hayes-compatible interface.

`MODEM.LIB` is not directly related to PPP. It allows ASCII strings to be sent to and received from the modem. Typically, these strings are AT commands and modem responses.

If you have special requirements for establishing communications with an ISP that cannot be handled by the default PPP library methods, you will need to explicitly include `MODEM.LIB` and write a program to establish the communications link. The program will typically need to command the modem to dial out; wait for a valid connection; send a user ID and password to the ISP and validate the response. After this has completed successfully, PPP can be started using the `ifup()` function.

For a complete description of `ifup()` and other Dynamic C TCP/IP functions, please see the *Dynamic C TCP/IP User's Manual, Vol. I*.

NOTE: `MODEM.LIB` is currently limited to controlling a single modem. The modem serial port and control lines are defined using macro constants, which should match with the definitions of the PPP interface.

The sample program `Samples\PPP\modem_test.c` shows the general idea for using `MODEM.LIB`.

7.3 Operation Details for PPP over Serial

The first step is to configure whatever transport medium will be used for the PPP connection. For directly connecting a serial line to the peer, the two serial data lines TX and RX may be adequate; however, the most common situation will be some sort of modem.

7.3.1 The Modem Interface

The interface between a modem and a controller is either a true RS232 interface or a variation on RS232 that uses TTL voltage levels for all of the signals. The latter are used by board-mounted modem modules. If an external modem is used, an RS232 transceiver chip is needed to convert RS232 voltages to logic signals and vice versa. A full RS232 connection has 3 outputs and 5 inputs from the controller's point of view.

In RS232 terminology, the controller is referred to as the DTE (Data Terminal Equipment). Modems and other peripherals are referred to as DCE's (Data Communications Equipment).

The specifics of a dial-up PPP connection are dependent on the modem hardware and the ISP. There are some settings that require information obtainable only from the ISP, like a phone number, a username, a password, etc.

The sample programs for use with a modem (\Samples\PPP\MODEM_*.c) define macros used by modem.lib. Contact your ISP to substitute the correct string values for any of these macros needed by your application:

```
#define DIALUP_NAME "username"
#define DIALUP_NUMBER "5551212"
#define DIALUP_PASSWORD "password"
#define EMAIL_FROM "rabbit@isp.com"
#define EMAIL_TO "you@wherever.com"
#define SMTP_SERVER "smtp.isp.com"
```

7.3.1.1 Rabbit Pin Connections to Modem

The modem control library, MODEM.LIB, defines default connections to the Rabbit as follows:

Table 7.1 Rabbit Pin Assignments for Modem Connection

RS232 Signal	Rabbit Pin	Direction
DTR	PB6	out
RTS	PB7	out
CTS	PB0	in
DCD	PB2	in
RI	PB3	in
DSR	PB4	in
TD	PC2	out
RD	PC3	in

7.3.2 Flow Control

Hardware flow control is implemented for the Rabbit PPP system. It follows the RS232 convention of using Ready To Send (RTS) and Clear To Send (CTS) lines.

Flow control is not required for speeds up to and including 115200 bps. The internal character processing is fast enough that the controller does not have to throttle incoming data flow. However, the modem or peer may need to throttle transmitted data. It is recommended that the RTS (modem to controller) line be connected for modems that cannot handle a continuous data stream at the specified rate. You can also connect the CTS (controller to modem) line, but the controller will merely assert this line continuously. To enable or disable hardware flow control, call `ifconfig()` with the `IFS_PPP_FLOWCONTROL` parameter identifier. You should also specify `IFS_PPP_RTSPIN` and `IFS_PPP_CTSPIN` parameter identifiers.

7.4 Operation Details for PPPoE

PPPoE avoids most of the complexities of PPP over serial. This is because the hardware (Ethernet) is easy to set up, and no modems are involved. Actually, you might have something called a DSL modem (or similar), but this type of modem does not have to do “dial-up” in the usual sense.

PPPoE is selected by defining the symbol `USE_PPPOE` to be a non-zero value. Currently, the only value supported is “1” with “2” reserved for future controller boards that have a second Ethernet chip. If you define `USE_PPPOE`, then you should also define `IFCONFIG_PPPOE0` to contain initialization options passed to `ifconfig()`. When PPPoE is specified, the interface is referred to by `IF_PPPOE0`. (`IF_PPPOE1` is reserved for future boards.)

7.5 Link Control Protocol Options

Link Control Protocol is the first sub-protocol used on a PPP link. The following LCP options are supported by the Rabbit PPP system:

Table 7.2 Configuration Options

LCP Configuration Option Type Field	Meaning of Option Type
01	MRU (Maximum-Receive-Unit)
02	ACCM (Async-Control-Character-Map)
03	Auth (Authentication-Type): PAP only
05	Magic Number
07	PFC (Protocol-Field-Compression)
08	ACFC (Address-and-Control-Field-Compression)

For more information on these options, refer to RFC 1661: The Point-to-Point Protocol (PPP) at, <http://www.faqs.org/rfcs/rfc1661.html>

7.6 Configuring PPP

Since multiple interfaces are supported, your application should call `ifconfig()` to change PPP interface parameters at run-time, or define suitable `IFCONFIG_PPP*` macros for boot-time configuration of each PPP interface (both serial and PPPoE).

You select serial port hardware to use with PPP by defining `USE_PPP_SERIAL` before including `dcrtcp.lib`. Similarly, you select PPPoE by defining `USE_PPPOE` before including `dcrtcp.lib`.

7.6.1 Serial Port Selection

PPP over asynchronous serial requires a suitable Rabbit serial port to be selected. You can use any of the available ports, since they all support asynchronous communications.

The serial port selection is entirely dynamic; however, there is a fixed mapping between interface numbers and serial port hardware. `IF_PPP0` always represents serial port A. `IF_PPP1` is always serial port B, and so on.

The serial port hardware to use is determined by the `USE_PPP_SERIAL` macro, which your application defines in order to specify PPP serial interfaces. `USE_PPP_SERIAL` is set to a bitwise OR combination of numbers representing the desired serial port(s). Ports are assigned according to the following table.

Table 7.3 Bitmap Values for `USE_PPP_SERIAL`

Interface Number	Serial Port	Bitmap Value
<code>IF_PPP0</code>	SERA	0x01
<code>IF_PPP1</code>	SERB	0x02
<code>IF_PPP2</code>	SERC	0x04
<code>IF_PPP3</code>	SERD	0x08
<code>IF_PPP4</code>	SERE	0x10
<code>IF_PPP5</code>	SERF	0x20

If multiple PPP serial interfaces are required, use (for example):

```
#define USE_PPP_SERIAL 0x0C
```

which, as the bitwise combination of 0x04 and 0x08, specifies `SERC` (`IF_PPP2`) and `SERD` (`IF_PPP3`).

7.6.2 PPPoE Port Selection

Since all Rabbit-based controller boards currently have at most a single Ethernet driver chip, only a single PPPoE interface is available (however, it can be shared with non-PPPoE Ethernet over the same hardware, that is, non-PPPoE Ethernet will use interface `IF_ETH0` while PPPoE will use `IF_PPPOE0`).

7.6.3 ifconfig() Options for PPP

The `ifconfig()` parameter identifiers described in this section pertain to any PPP interface, whether serial or Ethernet. There are a considerable number of options pertinent to PPP over asynchronous serial. PPPoE does not, as yet, require any special configuration options because of its relative simplicity.

The parameter identifiers listed here are passed to the `ifconfig()` function. They can also be used in the appropriate `IFCONFIG_PPP*` macro definitions, to ensure that the interface(s) are initialized correctly at boot time. For example, a run-time change to the userid and password might be coded as follows:

```
ifconfig (IF_PPP2,
          IFS_PPP_REMOTEAUTH, "myUserId", "myPassword",
          IFS_END);
```

The same definition, for boot-time initialization, might be coded as

```
#define IFCONFIG_PPP2 \
    other parameters \
    IFS_PPP_REMOTEAUTH, "myUserId", "myPassword", \
    other parameters
```

The general PPP properties set during initialization are:

Table 7.4 Macros for PPP Initialization (Serial and Ethernet)

Macro Name	Macro Description	Data Type(s) for Macro Params
<code>IFS_PPP_ACCEPTIP</code>	Accept peer's idea of our local IP address.	bool
<code>IFS_PPP_REMOTEIP</code>	Try to set IP address of peer.	longword
<code>IFS_PPP_ACCEPTDNS</code>	Accept a DNS server IP address from peer.	bool
<code>IFS_PPP_REMOTEDNS</code>	Set DNS server IP addresses for peer (primary and secondary).	longword, longword
<code>IFS_PPP_AUTHCALLBACK</code>	Called when a peer attempts to authenticate.	int (*)()
<code>IFS_PPP_INIT</code>	Sets up PPP with default parameters.	none
<code>IFS_PPP_REMOTEAUTH</code>	Sets username and password to give to peer.	char *, char *
<code>IFS_PPP_LOCALAUTH</code>	Required username and password for incoming peer	char *, char *

All of these `IFS_PPP_*` macros (except the initialization and callback) have `IFG_PPP_*` versions that allow an application to look at the current properties.

7.6.4 ifconfig() Options for Serial PPP

The `ifconfig()` parameter identifiers described in this section pertain to serial PPP interfaces only. (If you specify these options for PPPoE interfaces they will be quietly ignored.) They may also be specified in the appropriate `IFCONFIG_PPP*` macro definitions for boot-time initialization.

Table 7.5 Macros for PPP Initialization (for Serial)

Macro Name	Macro Description	Data Type(s) for Macro Params
<code>IFS_PPP_SPEED</code>	Set serial PPP speed (bps)	longword
<code>IFS_PPP_RTSPIN</code>	Define the RTS pin.	int, char *, int
<code>IFS_PPP_CTSPIN^f</code>	Define the CTS pin.	int, int
<code>IFS_PPP_USEPORTD</code>	Use parallel port D instead of parallel port C for serial ports A and B.	bool
<code>IFS_PPP_FLOWCONTROL</code>	Turn hardware flow control on or off	bool
<code>IFS_PPP_HANGUP</code>	An optional string to send to the modem after PPP shuts down.	char *
<code>IFS_PPP_MODEMESCAPE</code>	When enabled, sends modem escape sequences before send/expect or hangup sequence is: '<delay>+++<delay>' This is recognized by almost all modems to force them into command mode.	bool
<code>IFS_PPP_SENDEXPECT</code>	A formatted send and expect sequence for dialing and shell login.	char *
<code>IFS_PPP_USEMODEM</code>	Specify whether to use modem dialout string.	bool

All of these `IFS_PPP_*` macros have `IFG_PPP_*` versions that allow an application to look at the current properties.

The parameter for the `IFS_PPP_SENDEXPECT` option is a string containing a send/expect script to run when the PPP connection comes up. It is a series of tokens separated by spaces, alternating between a string to transmit, and a string to expect back.

For example:

```
SEscript = "ATDT5551212 CONNECT '' ogin: 'Joe User' word: secret PPP";
```

The sequence is:

1. Send ATDT5551212 - dials up an ISP.
2. Wait for the word CONNECT.
3. An empty send string, "" means don't send anything and wait for the next expect string.
4. Wait for "login:" or "Login:" By leaving off the 'L' either one will match.
5. Send 'Joe User'. Note that this token is contained in single quotes because it contains a space.
6. Wait for "password:" or "Password:"
7. Send the password.
8. Wait for the sequence 'PPP' This indicates a PPP session has started.

7.6.4.1 Additional Rules for Send/Expect Scripts

- A carriage return character (ASCII 13) is automatically sent after each send token.
- An ampersand (&) at the start of an expect token indicates that the driver should wait indefinitely for that token to be received. This is useful when waiting to answer a call, e.g., to set the modem to answer and wait indefinitely for a connection "ATSO=1 &CONNECT"
- As mentioned above, an empty token "" is immediately skipped. This allows for a chain of expect tokens to be used.
- The macro PSS_MODEM_CONNECT_WAIT determines the total time for the script. If this is exceeded, a timeout failure will occur and the interface will fail to come up. Using the ampersand modifier resets this timeout.

Note that the IFS_PPP_USEMODEM specifies that PPP assumes that it is talking to a modem. When the interface is being brought up, it will first run through the send/expect script. After the script completes, PPP will assume that it can launch straight into LCP. If this is not appropriate, do not use IFS_PPP_SENDEXPECT or IFS_PPP_USEMODEM. Instead, use the facilities of MODEM.LIB to perform an appropriate login to the ISP. Only when this is complete should you call `ifup()`.

Use of MODEM.LIB entails some limitations:

- Only one PPP serial interface can use MODEM.LIB.
- You need to configure MODEM.LIB to match the serial port you are using for PPP.
- Ensure that you specify an IFCONFIG_PPP* default such that the interface remains "down" at boot-time. In other words, do *not* append IFS_UP to the IFCONFIG_PPP* definition.

7.6.5 Starting and Stopping PPP Interfaces

The details of establishing and tearing down PPP links are handled by `sock_init()` and `tcp_tick()`, as are all other TCP/IP functions.

To start a PPP interface, `ifup()` is used, just as it is for non-PPPoE Ethernet interfaces. One difference that you should note is that the interface will not usually be up after `ifup()` returns. The function `ifup()` only sets the process in motion, which takes much longer for PPP than it does for non-PPPoE Ethernet.

Your application should be aware of this, since you will not be able to open sockets on an interface that is not fully enabled. If necessary, you can poll the interface to wait for it to come up. While polling, you *must* call `tcp_tick()` regularly. This is because it is actually the processing driven from `tcp_tick()` that drives the whole PPP negotiation machinery.

The correct way to poll an interface is given by the following code fragment. This code includes tests for the possibility that the interface may not be able to come up (e.g., because of a time-out).

```
ifup(IF_PPP2);
while (ifpending(IF_PPP2) == 1) tcp_tick();
if (!ifstatus(IF_PPP2))
    printf("Failed!\n");
```

A similar consideration applies for bringing the interface down:

```
ifdown(IF_PPP2);
while (ifpending(IF_PPP2) == 3) tcp_tick();
```

Note that there is no need to test for an interface “failing to come down,” however the tear-down process may take a short time. If you wait for the interface to come down before restarting it then there is a better chance that the link will come back up successfully, since the peer will have been notified properly.

NOTE: For PPP links with `IFS_PPP_USEMODEM` in effect, the process of bringing the interface up and down will include the modem dial-out and hang-up procedure. If you had `USEMODEM` in effect when connecting, but turned it off during the connection, then `ifdown()` will *not* perform modem hang-up. You will need to “manually” hang up the modem (or possibly just renegotiate from the LCP phase, if this is what you intended, by calling `ifup()`).

8. Function Reference

This section contains descriptions for all user-callable functions in `DCRTCP.LIB`. Starting with Dynamic C 7.05, `DCRTCP.LIB` is a light wrapper around:

- `DNS.LIB`
- `IP.LIB`
- `NET.LIB`
- `TCP.LIB`
- `UDP.LIB`.

This update requires no changes to existing code.

Descriptions for select user-callable functions in:

- `ARP.LIB`
- `BSDNAME.LIB`
- `ICMP.LIB`
- `IGMP.LIB`
- `PPP.LIB`
- `XMEM.LIB`

are also included here. Note that `ARP.LIB`, `ICMP.LIB` and `BSDNAME.LIB` are automatically `#use'd` from `DCRTCP.LIB`.

Functions are listed [alphabetically](#) and by [category](#) grouped by the task performed.

`_abort_socks`

```
int _abort_socks( byte reason, byte iface );
```

DESCRIPTION

Abort all open TCP and UDP sockets. This routine may be called if the network becomes unavailable, for example because a DHCP address lease expired or because an IP address conflict was encountered.

This function is generally intended for internal library use, but may be invoked by applications in special circumstances.

PARAMETERS

reason	Reason code. A suitable <code>NETERR_*</code> constant as defined in <code>NETERRNO.LIB</code> . This code is set as the error code for each socket that was affected.
iface	Specific interface on which active connections are to be aborted, or pass <code>IF_ANY</code> to abort connections on all active interfaces.

RETURN VALUE

0

LIBRARY

`net.lib`

SEE ALSO

[sock_abort](#), [sock_error](#)

arpcache_create

```
ATHandle arpcache_create( longword ipaddr );
```

DESCRIPTION

Create a new entry in the ARP cache table for the specified IP address. If a matching entry for that address already exists, then that entry is returned. Otherwise, a new entry is initialized and returned. If a new entry is created, then an old entry may need to be purged. If this is not possible, then `ATH_NOENTRIES` is returned.

PARAMETER

ipaddr IP address of entry.

RETURN VALUE

Positive value: Success.

`ATH_NOENTRIES`: No space is available in the table, and none of the entries could be purged because they were all marked as permanent or router entries.

LIBRARY

`ARP.LIB`

arpcache_flush

```
ATHandle arpcache_flush( ATHandle ath );
```

DESCRIPTION

Mark an ARP cache table entry for flushing. This means that the given table entry will be the first entry to be re-used for a different IP address, if necessary. Any entry (including permanent and router entries) may be flushed except for the broadcast entry.

PARAMETER

ath ARP table handle obtained from e.g., `arpcache_search()`.

RETURN VALUE

Positive value: Success.

ATH_UNUSED: The table entry was unused.

ATH_INVALID: the `ath` parameter was not a valid handle.

ATH_OBSOLETE: The given handle was valid, but obsoleted by a more recent entry. No change made.

LIBRARY

ARP.LIB

arpcache_hwa

```
ATHandle arpcache_hwa( ATHandle ath, byte *hwa );
```

DESCRIPTION

Copy the Ethernet (hardware) address from the given ARP cache table entry into the specified area.

PARAMETERS

ath	ARP cache table entry.
hwa	Address of where to store the hardware address (6 bytes).

RETURN VALUE

Positive value: Handle to the entry.
ATH_UNUSED: The table entry was unused.
ATH_INVALID: The ath parameter was not a valid handle.
ATH_OBSOLETE: The given handle was valid, but obsoleted by a more recent entry. No change made.

LIBRARY

ARP.LIB

arpcache_iface

```
ATHandle arpcache_iface( ATHandle ath, byte *iface );
```

DESCRIPTION

Copy the interface number from the given ARP cache table entry into the specified area.

If the `ath` parameter refers to a broadcast or loopback entry, then `*iface` is set to `IF_DEFAULT` (and `ATH_INVALID` is returned, since we can't really determine which of the interfaces to broadcast from).

PARAMETERS

ath	ARP cache table entry.
iface	Address of where to store the interface number (1 byte).

RETURN VALUE

Positive value: Handle to the entry.

`ATH_UNUSED`: The table entry was unused.

`ATH_INVALID`: The `ath` parameter was not a valid handle, or was a broadcast, multicast or loopback handle.

`ATH_OBSOLETE`: The given handle was valid, but obsoleted by a more recent entry.

LIBRARY

`ARP.LIB`

arpcache_ipaddr

```
ATHandle arpcache_ipaddr( ATHandle ath, longword *ipaddr );
```

DESCRIPTION

Copy the IP address from the given ARP cache table entry into the specified area. If the `ath` parameter refers to a broadcast entry, then the subnet broadcast IP is returned.

PARAMETERS

ath	ARP cache table entry.
ipaddr	Address of where to store the IP address (4 bytes).

RETURN VALUE

Positive value: Handle to the entry.

ATH_UNUSED: The table entry was unused.

ATH_INVALID: The `ath` parameter was not a valid handle, or was a point-point, broadcast, multicast or loopback handle.

ATH_OBSOLETE: The given handle was valid, but obsoleted by a more recent entry.

LIBRARY

ARP.LIB

arpcache_load

```
ATHandle arpcache_load( ATHandle ath, byte *hwa, byte iface,
                        word flags, byte router_used );
```

DESCRIPTION

Load an entry in the ARP cache table. The entry must have been created using `arpcache_create()`, or be an existing valid entry located via `arpcache_search()`.

This function is primarily intended for internal use by the ARP library, although advanced applications could also use it. Most applications should not need to call this function directly.

PARAMETERS

ath	Handle for the entry.
hwa	Hardware (Ethernet) address, or NULL. Pass NULL if the current hardware address is not to be changed.
iface	Interface to use (IF_DEFAULT to use default, or not change current setting).
flags	Flags for entry: one or more of the following values, OR'd together: <ul style="list-style-type: none">• ATE_PERMANENT: permanent entry• ATE_RESOLVING: initiate network resolve for this entry (hwa is ignored if this flag is set)• ATE_RESOLVED: this entry now resolved• ATE_ROUTER_ENT: this is a router entry• ATE_FLUSH: mark this entry for flush• ATE_VOLATILE: set short timeout for this entry• ATE_ROUTER_HOP: this entry uses the specified router as the first hop. hwa ignored.• ATE_REDIRECTED: this entry redirected by ICMP. <p>Only one of ATE_ROUTER_ENT or ATE_ROUTER_HOP should be set. For either of these, the next parameter indicates the router table entry to use.</p> <p>Only one of ATE_RESOLVING or ATE_RESOLVED should be set.</p>
router_used	Router table entry. Only used if one of ATE_ROUTER_ENT or ATE_ROUTER_HOP is set in the flags parameter.

`arpcache_load (cont.)`

RETURN VALUE

Positive value: Success.

`ATH_NOROUTER`: The specified router entry number is invalid. This can be because the `router_used` parameter is bad, or because the router entry has a mismatching ATH.

`ATH_INVALID`: Invalid table handle passed (or unused entry).

`ATH_OBSOLETE`: The given handle was valid, but obsoleted by a more recent entry. No change made.

LIBRARY

`ARP.LIB`

arpcache_search

```
ATHandle arpcache_search( longword ipaddr, int virt );
```

DESCRIPTION

Return handle that refers to the ARP cache table entry for the given IP address. This does not do any resolving. It only consults the existing cache entries. The returned handle is guaranteed to be valid at least until the next call to `tcp_tick()`. Usually the handle will be valid for considerably longer, however it is possible for the handle to become obsolete if the cache entry is re-used for a different address. The caller should be able to deal with this possibility. The entry returned for the broadcast address is guaranteed to be permanent.

PARAMETERS

ipaddr	IP address to locate in the cache. This may be -1L to locate the broadcast entry or our own IP address to return the "loopback" entry.
virt	0: Do not return the broadcast or loopback entries. 1: Allow the broadcast or loopback entries.

RETURN VALUE

Positive value: Handle to the entry.

ATH_NOTFOUND: No entry exists for the given IP address.

LIBRARY

ARP.LIB

arp_getArpData

```
ATEntry *arp_getArpData( void );
```

DESCRIPTION

Return address of `_arp_data[]`.

RETURN VALUE

address

LIBRARY

ARP.LIB

arp_getArpGateData

```
ATEntry *arp_getArpGateData( void );
```

DESCRIPTION

Return address of `_arp_gate_data[]`.

RETURN VALUE

address

LIBRARY

ARP.LIB

_arp_resolve

```
int _arp_resolve( longword ina, eth_address * ethap, int nowait );
```

DESCRIPTION

Gets the Ethernet address for the given IP address. This function is deprecated starting with Dynamic C 7.20.

PARAMETERS

ina	The IP address to resolve to an Ethernet address.
ethap	The buffer to hold the Ethernet address.
nowait	If 0, return within 750 ms; else if !0 wait up to 5 seconds trying to resolve the address.

RETURN VALUE

1: Success.
0: Failure.

LIBRARY

ARP.LIB

arpresolve_check

```
ATHandle arpresolve_check( ATHandle ath, longword ipaddr );
```

DESCRIPTION

Check up on status of resolve process initiated by `arpresolve_start()`. This function should be called regularly to ensure that an ARP table handle is pointing to the correct entry, and that the entry is still current.

This caller must call `tcp_tick()` if spinning on this function.

PARAMETERS

ath	ARP Table Handle obtained from <code>arpresolve_start()</code> .
ipaddr	IP address specified to <code>arpresolve_start()</code> . If this is zero, no check is performed. Otherwise, the ARP table entry is checked to see that it is the correct entry for the specified IP address.

RETURN VALUE

Positive value: Completed successfully. The return value will be the same as the `ath` parameter.

`ATH_AGAIN`: Not yet completed, try again later.

`ATH_FAILED`: Completed in error. Address cannot be resolved because of a network configuration problem.

`ATH_TIMEDOUT`: Resolve timed out. No response from addressee within the configured time limit.

`ATH_INVALID`: The `ath` parameter was not a valid handle.

`ATH_OBSOLETE`: The given handle was valid, but obsoleted by a more recent entry. Restart using `arpresolve_start()`.

`ATH_MISMATCH`: The `ipaddr` parameter was not zero, and the IP address does not match the table entry.

LIBRARY

`ARP.LIB`

arpresolve_ipaddr

```
longword arpsolve_ipaddr( AHandle ath );
```

DESCRIPTION

Given an ARP table handle, return the IP address of the corresponding table entry.

PARAMETER

ath ARP Table Handle obtained from e.g., `router_for()`.

RETURN VALUE

0: An error occurred, such as an invalid or obsolete handle.

0xFFFFFFFF: The handle refers to either the broadcast address, or to a point-to-point entry whose IP address is not defined.

Else: An IP address. This may be 127.0.0.1 for the loopback entry.

LIBRARY

ARP.LIB

arpresolve_start

```
ATHandle arpresolve_start( longword ipaddr );
```

DESCRIPTION

Start resolve process for the given IP address. This may return immediately if the IP address is in the ARP cache table and still valid. Otherwise, if the IP address is on the local subnet then an ARP resolve request is issued through the appropriate interface. If the address is not on the local subnet, then a router table entry is used and no network activity is necessary (unless the router itself is not resolved, in which case its resolution commences).

PARAMETER

ipaddr IP address of host whose hardware address is to be resolved.

RETURN VALUE

Positive value: Success. The value is actually the ATH of the ARP cache table entry which is (or will be) used. This value should be passed to subsequent calls to `arpresolve_check()`.

ATH_NOENTRIES: No space is available in the table, and none of the entries could be purged, because they were all marked as permanent or router entries.

ATH_NOROUTER: No router ("gateway") is configured for the specified address, which is not on the local subnet.

LIBRARY

ARP.LIB

arpresolve_start_iface

```
ATHandle arpresolve_start_iface( longword ipaddr, word iface );
```

DESCRIPTION

Start resolve process for the given IP address. This may return immediately if the IP address is in the ARP cache table and still valid. Otherwise, if the IP address is on the local subnet of the specified interface then an ARP resolve request is issued through that interface. If the address is not on the local subnet, then a router table entry (with router on the given interface) is used and no network activity is necessary (unless the router itself is not resolved, in which case its resolution is commenced).

If `ipaddr` is zero, then the broadcast entry is returned, since this is required for DHCP when our IP address is not yet known.

This function is identical to `arpresolve_start()` if the `iface` parameter is set to `IF_ANY`.

PARAMETERS

ipaddr	IP address of host whose hardware address is to be resolved.
iface	Specific interface (or <code>IF_ANY</code> to get the library to select the most appropriate interface). A specific interface forces the use of that interface, even if there is no route available. This function should only be used to override the normal interface selection procedure.

RETURN VALUE

Positive value: Success. This value is actually the ATH of the ARP cache table entry which is (or will be) used; it should be passed to subsequent calls to `arpresolve_check()`.

`ATH_NOENTRIES`: No space is available in the table, and none of the entries could be purged, because they were all marked as permanent or router entries.

`ATH_NOROUTER`: No router (“gateway”) is configured for the specified address, which is not on the local subnet of the specified interface.

LIBRARY

`ARP.LIB`

SEE ALSO

[arpresolve_start](#)

`_arp_send_gratuitous`

```
int _arp_send_gratuitous( word iface );
```

DESCRIPTION

When a host sends an ARP request to resolve its own IP address, it is called gratuitous ARP. In the ARP request packet, the source IP address and destination IP address are filled with the same source IP address itself. The destination MAC address is the Ethernet broadcast address (FF:FF:FF:FF:FF:FF).

Gratuitous ARP is used for the following:

1. In a properly configured network, there will not be an ARP reply for a gratuitous ARP request. But if another host in the network is also configured with the same IP address as the source host, then the source host will get an ARP reply. In this way, a host can determine whether another host is also configured with its IP address.
2. When the network interface card in a system is changed, the MAC address to its IP address mapping is changed. In this case, when the host is rebooted, it will send an ARP request packet for its own IP address. As this is a broadcast packet, all the hosts in the network will receive and process this packet. They will update their old mapping in the ARP cache with this new mapping.

PARAMETER

<code>iface</code>	Interface number.
--------------------	-------------------

RETURN VALUE

1: Success
0: Failure

LIBRARY

`ARP.LIB`

aton

```
longword aton( char *text );
```

DESCRIPTION

Converts [a.b.c.d] or a.b.c.d to a 32 bit long value.

PARAMETER

text Pointer to string that holds the IP address to convert.

RETURN VALUE

0: Error, string has invalid format.
>0: Success, long value of IP address.

LIBRARY

IP.LIB

`_chk_ping`

```
longword _chk_ping( longword host_ip, longword *sequence_number );
```

DESCRIPTION

Checks for any outstanding ping replies from host. `_chk_ping` should be called frequently with a host IP address. If an appropriate packet is found from that host IP address, the sequence number is returned through `*sequence_number`. The time difference between our request and their response is returned in milliseconds.

PARAMETERS

`host_ip` IP address to receive ping reply from.

`sequence_number` Sequence number of reply.

RETURN VALUE

Time in milliseconds from the ping request to the host's ping reply.

If `_chk_ping` returns `0xffffffffL`, there were no ping receipts on this current call.

LIBRARY

`ICMP.LIB`

SEE ALSO

`_ping`, `_send_ping`

dhcp_acquire

```
int dhcp_acquire( void );
```

DESCRIPTION

This function acquires a DHCP lease that has not yet been obtained, or has expired, or was relinquished using `dhcp_release()`. Normally, DHCP leases are renewed automatically, however if the DHCP server is down for an extended period then it might not be possible to renew the lease in time, in which case the lease expires and TCP/IP should not be used. When the lease expires, `tcp_tick()` will return 0, and the global variable for the IP address will be reset to 0. At some later time, this function can be called to try to obtain an IP address.

This function blocks until the lease is renewed, or the process times out.

RETURN VALUE

- 0: OK, lease was not expired, or an IP address lease was acquired with the same IP address as previously obtained.
- 1: An error occurred, no IP address is available. TCP/IP functionality is thus not available. Usual causes of an error are timeouts because a DHCP or BOOTP server is not available within the timeout specified by the global variable `_bootptimeout` (default 30 seconds).
- 1: Lease was re-acquired, however the IP address differs from the one previously obtained. All existing sockets must be re-opened. Normally, DHCP servers are careful to reassign the same IP address previously used by the client, however this is sometimes not possible.

LIBRARY

`BOOTP.LIB`

dhcp_get_timezone

```
int dhcp_get_timezone( long *seconds );
```

DESCRIPTION

This function returns the time zone offset provided by the DHCP server, if any, or uses the fall-back time zone defined by the `TIMEZONE` macro. Note that `TIMEZONE` is expressed in hours, whereas the return result is in seconds.

PARAMETERS

seconds	Pointer to result longword. If the return value is 0 (OK), then this will be set to the number of seconds offset from Coordinated Universal Time (UTC). The value will be negative for west; positive for east of Greenwich. If the return value is -1, then the result will be set using the hard-coded value from the macro <code>TIMEZONE</code> (converted to seconds by multiplying by 3600), or zero if this macro is not defined.
----------------	--

RETURN VALUE

0: Time zone obtained from DHCP.
-1: Time zone not valid, or not yet obtained, or not using DHCP.

LIBRARY

`BOOTP.LIB`

dhcp_release

```
int dhcp_release( void );
```

DESCRIPTION

This function relinquishes a lease obtained from a DHCP server. This allows the server to re-use the IP address that was allocated to this target. After calling this function, the global variable for the IP address is set to 0, and it is not possible to call any other TCP/IP function which requires a valid IP address. Normally, `dhcp_release()` would be used on networks where only a small number of IP addresses are available, but there are a large number of hosts which need sporadic network access.

This function is non-blocking since it only sends one packet to the DHCP server and expects no response.

RETURN VALUE

- 0: OK, lease was relinquished.
- 1: Not released, because an address is currently being acquired, or because a boot file (from the BOOTP or DHCP server) is being downloaded, or because some other network resource is in use e.g., open TCP socket. Call `dhcp_release()` again after the resource is freed.
- 1: Not released, because DHCP was not used to obtain a lease, or no lease was acquired.

LIBRARY

BOOTP.LIB

getdomainname

```
char *getdomainname( char *name, int length );
```

DESCRIPTION

Gets the current domain name. For example, if the controller's internet address is "test.mynetwork.com" then "mynetwork" is the domain portion of the name.

The domain name can be changed by the `setdomainname()` function.

PARAMETERS

name	Buffer to place the name.
length	Maximum length of the name, or zero to get a pointer to the internal domain name string. Do not modify this string!

RETURN VALUE

If `length ≥ 1`: Pointer to name. If `length` is not long enough to hold the domain name, a NULL string is written to name.

If `length = 0`: Pointer to internal string containing the domain name. Do not modify this string!

LIBRARY

BSDNAME.LIB

SEE ALSO

[setdomainname](#), [gethostname](#), [sethostname](#), [getpeername](#), [getsockname](#)

EXAMPLE

```
main() {
    sock_init();
    printf("Using %s for a domain\n", getdomainname(NULL, 0));
}
```

gethostid

```
longword gethostid( void );
```

DESCRIPTION

Return the IP address of the controller in host format.

RETURN VALUE

IP address in host format, or zero if not assigned or not valid.

LIBRARY

IP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sethostid](#)

EXAMPLE

```
main() {
    char buffer[512];
    sock_init();
    printf("My IP address is %s\n", inet_ntoa( buffer, gethostid()));
}
```

gethostname

```
char *gethostname( char *name, int length );
```

DESCRIPTION

Gets the host portion of our name. For example if the controller's internet address is "test.mynetwork.com" the host portion of the name would be "test."

The host name can be changed by the `sethostname()` function.

PARAMETERS

name	Buffer to place the name.
length	Maximum length of the name, or zero for the internal host name buffer. Do not modify this buffer.

RETURN VALUE

`length ≥ 1`: Return name.

`length = 0`: Return internal host name buffer (do not modify!).

LIBRARY

`BSDNAME.LIB`

getpeername

```
int getpeername( sock_type *s, void *dest, int *len );
```

DESCRIPTION

Gets the peer's IP address and port information for the specified socket.

PARAMETERS

s	Pointer to the socket.
dest	Pointer to <code>sockaddr</code> to hold the socket information for the remote end of the socket. The data structure is: <pre>typedef struct sockaddr { word s_type; // reserved word s_port; // port #, or 0 if not connected longword s_ip; // IP addr, or 0 if not connected byte s_spares[6]; // not used for tcp/ip connections };</pre>
len	Pointer to the length of <code>sockaddr</code> . A NULL pointer can be used to represent the <code>sizeof(struct sockaddr)</code> .

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

BSDNAME.LIB

SEE ALSO

[getsockname](#)

getsockname

```
int getsockname( sock_type *s, void *dest, int *len );
```

DESCRIPTION

Gets the controller's IP address and port information for a particular socket.

PARAMETERS

s	Pointer to the socket.
dest	Pointer to <code>sockaddr</code> to hold the socket information for the local end of the socket. The data structure is: <pre>typedef struct sockaddr { word s_type; // reserved word s_port; // port #, or 0 if not connected longword s_ip; // IP addr, or 0 if not connected byte s_spares[6]; // not used for tcp/ip connections };</pre>
len	Pointer to the length of <code>sockaddr</code> . A NULL pointer can be used to represent the <code>sizeof(struct sockaddr)</code> . <code>BSDNAME.LIB</code> will assume 14 bytes if a NULL pointer is passed.

RETURN VALUE

0: Success.
-1: Failure.

LIBRARY

`BSDNAME.LIB`

SEE ALSO

[getpeername](#)

htonl

```
longword htonl( longword value );
```

DESCRIPTION

This function converts a host-ordered double word to a network-ordered double word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network-byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order.

PARAMETERS

value Host-ordered double word.

RETURN VALUE

Host word in network format, e.g., `htonl(0x44332211)` returns `0x11223344`.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[htons](#), [ntohl](#), [ntohs](#)

hton

```
word hton( word value );
```

DESCRIPTION

Converts host-ordered word to a network-ordered word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network-byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order within each 16-bit section.

PARAMETERS

value Host-ordered word.

RETURN VALUE

Host-ordered word in network-ordered format, e.g., `hton(0x1122)` returns `0x2211`.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[htonl](#), [ntohl](#), [ntohs](#)

ifconfig

```
int ifconfig( int iface,... );
```

DESCRIPTION

This function sets and gets network parameters at runtime. An arbitrary number of parameters may be set or retrieved in one call.

This function replaces `tcp_config()` for setting network parameters at runtime. In addition, it allows retrieval of parameters and supports multiple interfaces. An arbitrary number of parameters may be set or retrieved in one call.

Example:

```
ifconfig( IF_ETH0,
          IFS_DOWN,
          IFS_IPADDR, aton("10.10.6.100"),
          IFS_NETMASK, 0xFFFFFFFF0uL,
          IFS_ROUTER_SET, aton("10.10.6.1"),
          IFS_NAMESERVER_SET, aton("192.68.1.123"),
          IFS_NAMESERVER_ADD, aton("192.68.1.124"),
          IFS_UP,
          IFS_END);
```

This call to `ifconfig()` brings the first Ethernet interface down if it is not already inactive, then it configures the home IP address, netmask, router (gateway), and two nameservers. Then, the interface is made active (`IFS_UP`). `IFS_END` is required to terminate the parameter list.

Debugging note: if you experience an error dialog indicating “bad parameter passed to I/O function,” this is often caused by this function encountering a bad parameter. In this case, it is easier to debug if you temporarily `#define NET_VERBOSE` at the top of your application code. This will cause this function to print some more details about the parameter that it found to be in error. The parameter numbers refer to IFS/IFG parameter identifiers, not to the sub-parameters of the identifiers. In the above example, `IFS_DOWN` would be considered the first parameter, and `IFS_UP` would be the 7th (not the 12th).

Most errors are caused by:

- passing an `int` when a `long` is expected, so be sure to add the appropriate cast in your code.
- forgetting to terminate the parameter list with `IFS_END`.

ifconfig (cont'd)

PARAMETERS

iface Interface number. Use one of the definitions:

- IF_DEFAULT
- IF_ETH0
- IF_ETH1
- IF_WIFI0
- IF_PPPOE0
- IF_PPPOE1
- IF_PPP0-5
- IF_ANY

If the interface does not exist, then you will get a compile time error. IF_DEFAULT refers to the default interface, which will be equivalent to the first of the other interface identifiers in the above list which exists. IF_ANY may be used only for the parameters which are not specific to any particular interface. It can also be used, where applicable, to mean “all interfaces” if the operation would make sense when applied to all interfaces.

...

Parameters 2 through n are polymorphic (like `printf()` parameters). Parameters are provided in groups (usually pairs) with the first parameter in the group being one of a documented set of identifiers, and any subsequent parameters in the group being the value specific to that identifier. The data type for “bool” parameter really means an integer, whose value is 0 for false, or non-zero for true.

PARAMETER IDENTIFIERS FOR IFCONFIG()

IFS_END

Marks the end of the parameter list. The list of parameter groups **MUST** be terminated using the identifier `IFS_END`.

IFS_UP

Bring up interface.

IFS_DOWN

Bring down interface.

ifconfig (cont'd)

IFS_IPADDR

Set home IP address (longword).

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

The action of `IFS_IPADDR` depends on the current interface state. If the i/f has the `IFS_DHCP` flag set, then this parameter sets only the fallback IP address without changing the current i/f status. Otherwise, the i/f is reconfigured with the new address immediately, which may require it to be brought down then up. `IFS_IPADDR` always sets the DHCP fallback address, but you can also use the `IFS_DHCP_FB_IPADDR` parameter to set the fallback address without ever changing the i/f status.

IFG_IPADDR

Get home IP address (longword *).

IFS_NETMASK

Set netmask (longword).

IFG_NETMASK

Get netmask (longword *).

IFS_MTU

Set maximum transmit unit (MTU) (word).

IFG_MTU

Get MTU (word *).

IFS_ROUTER_SET

Delete all routers, then set this one as a default router (longword).

This parameter does not care about the value of “iface” because it is not specific to an interface.

ifconfig (cont'd)

IFS_ROUTER_SET_STATIC

Set static router:

IP address of router, longword
subnet served, longword
subnet mask, longword

“Static router” means a router that handles routing to a specified subnet destination. When a router is selected for a given IP address, the most specific static router will be used. For example, given the following setup:

<u>Router</u>	<u>Subnet</u>	<u>Mask</u>
10.10.6.1	0	0
10.10.6.2	10.99.0.0	255.255.0.0
10.10.6.3	10.99.57.0	255.255.255.0

Then, given a destination IP address (which is not on the local subnet 10.10.6.0), the router will be selected according to the following algorithm:

if address is 10.99.57.*, use 10.10.6.3
else if address is 10.99.*.*, use 10.10.6.2
else use 10.10.6.1

Note that `IFS_ROUTER_SET` is basically the same as `IFS_ROUTER_SET_STATIC`, except that the subnet and mask parameters are automatically set to zero. Most simple networks with a single router to non-local subnets will use a single `IFS_ROUTER_SET`.

This parameter does not care about the value of “iface” because it is not specific to an interface.

IFS_ROUTER_ADD

Add general router (longword).

This parameter identifier does not care about the value of “iface” because it is not specific to an interface.

IFS_ROUTER_ADD_STATIC

Add static router:

IP address of router (longword)
subnet served (longword)
subnet mask (longword)

See [IFS_ROUTER_SET_STATIC](#) for a definition of static router.

This parameter does not care about the value of “iface” because it is not specific to an interface.

ifconfig (cont'd)

IFS_ROUTER_DEL

Delete router (longword). If identifier's parameter = 0, delete all routers.

This parameter does not care about the value of "iface" because it is not specific to an interface.

IFG_ROUTER_DEFAULT

Get default router (longword *).

The interface parameter may be either a specific interface number (to get the default router for that interface), or IF_ANY which will retrieve an overall default router.

IFS_HWA

Set the hardware address (byte *).

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

IFG_HWA

Get the hardware address (byte *).

IFS_NAMESERVER_SET

Delete all nameservers, then set this one (longword).

This parameter does not care about the value of "iface" because it is not specific to an interface.

IFS_NAMESERVER_ADD

Add nameserver (longword).

This parameter does not care about the value of "iface" because it is not specific to an interface.

IFS_NAMESERVER_DEL

Delete nameserver (longword).

This parameter does not care about the value of "iface" because it is not specific to an interface.

IFS_ICMP_CONFIG

Allow "arp -s" ping to configure IP address, (bool).

If DHCP and ping configure are both set, then the completion of DHCP will automatically turn off ping configure. If DHCP fails, then ping configure will be allowed after the set time-out for DHCP. Ping config cannot override DHCP until DHCP has timed out. This is the case whenever a DHCP lease is obtained, whether or not at `sock_init()` time.

This parameter may be set for IF_ANY i.e., all interfaces.

ifconfig (cont'd)

IFG_ICMP_CONFIG

Is ping configure enabled? (bool *)

IFG_ICMP_CONFIG_OK

Was ping configured successfully? (bool *)

IFS_ICMP_CONFIG_RESET

After ping configured okay, allow new ping configure.

IFS_DEBUG

Set debug level (int).

Sets a global variable, `debug_on`: 0 to print only a few messages, up to 5 to print successively more messages. This parameter does not care about the value of “iface” because it is not specific to an interface.

IFG_DEBUG

Get debug level (int *).

This parameter does not care about the value of “iface” because it is not specific to an interface.

IFS_IF_CALLBACK

Set interface up/down callback, or NULL (void (*)()).

The interface up/down callback function is called with two parameters:

```
ifcallback( int iface, int up);
```

where “iface” is the interface number, and “up” is non-zero if the interface has just come up, or zero if it has just come down. You must `#define USE_IF_CALLBACK` before `#use "dcrtcp.lib"` to use this functionality.

PARAMETER IDENTIFIERS FOR DHCP

The DHCP parameters are only available if `USE_DHCP` is defined, and will only work if the interface is qualified for DHCP. The `IFS_DHCP` parameter will cause acquisition or release of the specified interface.

IFS_DHCP

Use DHCP to configure this interface (bool = 0 for false, non-zero for true).

IFG_DHCP

Get DHCP setting (bool *).

IFG_DHCP_OK

Get whether DHCP actually configured OK (bool *).

IFS_DHCP_TIMEOUT

Set DHCP overall timeout in seconds (int).

ifconfig (cont'd)

IFG_DHCP_TIMEOUT

Get DHCP overall timeout in seconds (int *).

IFS_DHCP_QUERY

Set whether DHCP uses INFORM (bool). This parameter specifies that DHCP INFORM message is used for Ethernet interfaces, and is applicable if the IP address is configured other than by DHCP. The parameter is always TRUE for PPP interfaces.

IFG_DHCP_QUERY

Get whether DHCP uses INFORM (bool).

IFS_DHCP_DOMAIN

Set whether to use domain and/or hostname information (bool).

IFG_DHCP_DOMAIN

Get flag setting (bool *).

IFS_DHCP_FALLBACK

Set whether DHCP allows fallback to static configuration (bool).

IFG_DHCP_FALLBACK

Does DHCP allow fallback to static configuration? (bool *).

IFS_DHCP_FB_IPADDR

Set the DHCP fallback IP address (longword).

The DHCP fallback address parameters are used in preference to IFS_IPADDR (the “current” address). This indicates the static IP address to use in case DHCP could not be used to configure the interface.

The action of IFS_IPADDR depends on the current interface state. If the i/f has the IFS_DHCP flag set, then this parameter sets only the fallback IP address without changing the current i/f status. Otherwise, the i/f is reconfigured with the new address immediately, which may require it to be brought down then up. IFS_IPADDR always sets the DHCP fallback address, but you can also use the IFS_DHCP_FB_IPADDR parameter to set the fallback address without ever changing the i/f status.

IFG_DHCP_FB_IPADDR

Get the DHCP fallback IP address (longword *). See the description above for IFS_DHCP_FB_IPADDR for more information.

IFG_DHCP_FALLBACK

Get whether DHCP actually had to use fallbacks (bool *).

ifconfig (cont'd)

IFS_DHCP_OPTIONS

Set DHCP custom options (int, char *, int(*)()).

DHCP custom options processing: First parameter (int) is length of options list. The second parameter (char *) points to that options list. This is a byte array containing values from the DHCP_VN_* definitions in BOOTP.LIB (these are taken from the list in RFC2132). Also, option "0" is used to indicate the boot file name. If the boot file name is provided, then the TFTP server IP address can be obtained from the di->bootp_host field of the structure provided to the callback (see below). This options list must be in static storage, since only the pointer is saved.

The third parameter may be NULL, or is a pointer to a callback function to process the custom options. The callback function has the following prototype:

```
int my_callback (int iface, DHCPInfo *di, int opt, int len,
char * data)
```

iface: interface number. di: DHCP information struct. Read only, except you can modify the 'data' field if desired. See the definition of this struct in NET.LIB for details.

opt: DHCP option number (DHCP_VN_*); or 0 for the boot file name.

len: length of option data in bytes

data: pointer to data for this option. Read only.

The callback is only invoked for options which were requested and which were not handled internally (such as DHCP_VN_SUBNET). The return value from the callback should be zero, for future compatibility. The callback should not make any long computations, blocking calls, or call any other TCP/IP functions, since it would delay the main application. If uC/OS is in use, it should also be re-entrant and definitely not call any tcp/ip functions. Note that the following options are always retrieved and MUST NOT be provided in the options list:

- All DHCP protocol options (50-61)
- DHCP_VN_SUBNET
- DHCP_VN_TIMEOFF

The other options are only forbidden if DHCP_NUM_ROUTERS, etc., is defined non-zero:

- DHCP_VN_ROUTER
- DHCP_VN_DNS
- DHCP_VN_SMTPSRV
- DHCP_VN_NTPSRV
- DHCP_VN_COOKIE

IFG_DHCP_OPTIONS

Get DHCP custom options (int *, char **).

IFG_DHCP_INFO

Get DHCP information, or NULL if not qualified (DHCPInfo **).

ifconfig (cont'd)

PARAMETER IDENTIFIERS FOR PPP

The following parameter identifiers are for PPP interfaces only. You will get a runtime error (non-zero return code) if you apply one of the PPP-specific parameters to a non-PPP interface.

IFS_PPP_INIT

Sets up PPP with default parameters. This should be used before any other PPP setting parameters.

IFS_PPP_SPEED

Set serial PPP speed in bits per second (longword).

IFG_PPP_SPEED

Get serial PPP speed (longword).

IFS_PPP_ACCEPTIP

Accept peer's idea of our local IP address (bool).

IFG_PPP_ACCEPTIP

Are we accepting peer's idea of our local IP address? (bool *)

IFS_PPP_REMOTEIP

Try to set peer's IP address (longword).

IFG_PPP_REMOTEIP

Will we try to set peer's IP address? (longword *)

IFS_PPP_ACCEPTDNS

Accept a DNS server IP address from peer (bool).

IFG_PPP_ACCEPTDNS

Will we accept a DNS server IP address from peer? (bool *)

IFS_PPP_REMOTEDNS

Set DNS server IP addresses for peer; primary (longword), secondary (longword).

IFG_PPP_REMOTEDNS

Get DNS server IP addresses; primary (longword *) and secondary (longword *).

IFS_PPP_AUTHCALLBACK

Called when a peer attempts to authenticate (int (*)()).

The authentication callback is invoked with the following parameters:

int auth_cb (char *user, int userlen, char *pass, int passlen)

The parameters indicate userid, password and their lengths (not null terminated). The callback should return 1 if OK, 0 if not authorized.

ifconfig (cont'd)

IFS_PPP_REMOTEAUTH

Sets username and password to give to peer (char *, char *).

IFG_PPP_REMOTEAUTH

Get username and password given to peer (char **, char **).

IFS_PPP_LOCALAUTH

Required username and password for incoming peer
(char *, char *).

IFG_PPP_LOCALAUTH

char **, char **

IFS_PPP_RTSPIN

Define the RTS pin (int, char *, int).

The parameters for the RTS/CTS pin assignments are:

RTS: int port_address, char * shadow_reg, int port_pin

CTS: int port_address, int port_pin

where "port_address" is the parallel port internal I/O address e.g., PEDR for port E.

"shadow_reg" is the appropriate shadow register for the parallel port data register e.g.

&PEDRShadow for port E. "port_pin" is a number from 0-7 indicating the pin number of the port.

IFG_PPP_RTSPIN

Get RTS pin definitions (int *, char **, int *).

IFS_PPP_CTSPIN

Define the CTS pin (int, int). See description for [IFS_PPP_RTSPIN](#).

IFG_PPP_CTSPIN

Get CTS pin definitions (int *, int *).

IFS_PPP_FLOWCONTROL

Turn hardware flow control on or off (bool).

IFG_PPP_FLOWCONTROL

Get hardware flow control setting (bool *).

IFS_PPP_USEPORTD

Use parallel port D instead of parallel port C for serial ports A & B (bool).

IFG_PPP_USEPORTD

Are we using parallel port D instead of C? (bool *)

ifconfig (cont'd)

IFS_PPP_USEPORTE

Use parallel port E instead of parallel port C for serial ports E & F (bool). (Rabbit 4000 only)

IFG_PPP_USEPORTE

Are we using parallel port E instead of C? (bool *)

IFG_PPP_PEERADDR

Get the PPP peer address. Returns 0 if no connection (longword *).

IFS_PPP_PASSIVE

Set passive mode for PPP (bool). If TRUE, then interface will wait indefinitely for a connection after its initial connection attempt.

IFG_PPP_PASSIVE

Is passive mode set for PPP? (bool *)

THE FOLLOWING PARAMETER IDENTIFIERS ARE FOR MODEMS ON PPP INTERFACES:**IFS_PPP_SENDEXPECT**

A series of strings to send and then expect, each separated by a carriage return (“\r”) (char *).

Setting send/expect automatically turns on IFS_PPP_USEMODEM. See the documentation for `chat_init()` for details on the syntax for these strings. The specified strings MUST be in static storage, since only the pointers are stored rather than copying the strings.

Note that two substitution parameters are available: %0 may be used to insert the current user name (as set using the IFS_PPP_REMOTEAUTH command) and %1 is the corresponding password. This is useful if your logon script uses the same userid/password as is used by PPP during its authentication (PAP) stage.

IFG_PPP_SENDEXPECT

char **

IFS_PPP_USEMODEM

Specify whether to use modem dialout string (bool).

IFG_PPP_USEMODEM

Is modem dialout string going to be used? (bool *)

IFS_PPP_MODEMESCAPE

Specify whether or not to add an escape sequence <delay>+++<delay> before sending send/expect or hangup strings (bool).

ifconfig (cont'd)

IFG_PPP_MODEMESCAPE

Will escape sequence <delay>+++<delay> be added before sending send/expect or hangup strings? (bool *)

IFS_PPP_HANGUP

Optional string to send to modem to shut it down, in send-expect format. See [IFS_PPP_SENDEXPECT](#) for more information.

IFG_PPP_HANGUP

Get optional string (char **).

PARAMETER IDENTIFIERS FOR DMA

The following two parameter identifiers rely on the existence of DMA channels, thus are only for PPP or VSPD interfaces on Rabbit 4000 or newer Rabbit processors:

IFS_USE_DMA

Use the specified DMA channel instead of the serial port.

- `dma_chan_t`: specifies channel for source
- `dma_chan_t`: specifies channel for destination
- `word`: specifies I/O port address

IFS_USE_SERIAL

Use the serial port directly. This undoes the effect of `IFS_USE_DMA`.

Any PPP over asynchronous serial, or VSPD (Virtual Stream Packet Driver) interface may be switched between its normal fixed async serial port and a pair of DMA channels allocated to the application via the `DMAalloc()` function in `DMA.LIB`. The DMA channels should not be modified by the application while in use by the network library.

When specifying DMA channels (type `dma_chan_t`), you also provide a third, `word` parameter that specifies the I/O register to be the source and destination for incoming and outgoing data respectively. This does not need to be a serial port data register, but it usually would be. Currently, only internal I/O registers are supported for this function.

The application is responsible for setting serial port parameters as desired. When using DMA channels, the network library routines do not make any changes to or make any assumptions about the underlying I/O port. The same applies to the DMA channel. For example, the application is responsible for enabling the external DMA request line if, for example, it is desired to implement hardware transmit flow control.

Use of a DMA channel reduces the amount of CPU overhead taken up servicing interrupts. This allows higher communication rates, or allows the application to perform more useful work in the same time.

If the DMA channels are released via `IFS_USE_SERIAL` then it is the application's responsibility to re-use or unallocate the DMA channels as desired.

ifconfig (cont'd)

PARAMETER IDENTIFIERS FOR WI-FI INTERFACES

The following parameter identifiers are only for Wi-Fi interfaces. You will get a runtime error (non-zero return code) if you apply one of the Wi-Fi-specific parameters to a non-Wi-Fi interface. See the “Wi-Fi Configuration” documentation in `tcp_config.lib` for details on configuring the Wi-Fi interface.

IFS_WIFI_SSID

Set the SSID for the Wi-Fi device (int, byte *).

Since the SSID can contain any byte (including nulls), it's necessary to provide the length along with the SSID. See `wifi_ssid_to_str()` for creating a null-terminated, printable version of the SSID, with nulls and non-printable characters (byte values 0x00-0x20 and 0x7F-0xFF) replaced with “?”.

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

IFG_WIFI_SSID

Get the currently configured SSID (int *, byte *). See the `set` command for more information on the SSID.

IFS_WIFI_MULTI_DOMAIN

Enable multi-domain (bool). This only works on APs with 802.11d.

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

IFG_WIFI_MULTI_DOMAIN

Is multi-domain enabled? (bool *).

ifconfig (cont'd)

IFS_WIFI_REGION

Set regulatory region (int). Valid parameters and the channels they allow are:

```
IFPARAM_WIFI_REGION_AMERICAS, 1-11
IFPARAM_WIFI_REGION_AUSTRALIA, 1-11
IFPARAM_WIFI_REGION_CANADA, 1-11
IFPARAM_WIFI_REGION_CHINA, 1-11
IFPARAM_WIFI_REGION_EMEA, 1-13
IFPARAM_WIFI_REGION_FRANCE, 10-13
IFPARAM_WIFI_REGION_ISRAEL, 3-11
IFPARAM_WIFI_REGION_JAPAN, 1-13
IFPARAM_WIFI_REGION_MEXICO_INDOORS, 1-11
IFPARAM_WIFI_REGION_MEXICO_OUTDOORS, 9-11
```

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

IFG_WIFI_REGION

Get region number (int *). See set command for more information.

IFG_WIFI_REGION_INFO

Get region settings (wifi_region *).

The `wifi_region` structure contains the following elements:

```
typedef struct {
    char id;                // IFPARAM_WIFI_REGION_*
    char country[16];      // description of region
    int first_channel;
    int last_channel;
    unsigned int channel_mask;
    int max_pwr_dBm;
    int max_pwr_index;
} wifi_region;
```

IFS_WIFI_MODE

Set operating mode (int). Valid parameters are:

- IFPARAM_WIFI_ADHOC
- IFPARAM_WIFI_INFRASTRUCTURE

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

ifconfig (cont'd)

IFG_WIFI_MODE

Get operating mode (int *).

IFS_WIFI_CHANNEL

Set channel (int). See [IFS_WIFI_REGION](#) for more information about available channels. If using infrastructure mode can be set to “0” for automatic channel selection.

Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

IFG_WIFI_CHANNEL

Get current channel setting (int *).

IFS_WIFI_ENCRYPTION

Set encryption (word). Setting the value of this parameter may require the interface to be brought down temporarily. If this is necessary, it will be brought up again before return; however, any sockets that were open on that interface will have been aborted.

Valid values are:

- IFPARAM_WIFI_ENCR_ANY
- IFPARAM_WIFI_ENCR_NONE
- IFPARAM_WIFI_ENCR_WEP

If `WIFI_USE_WPA` is defined:

- IFPARAM_WIFI_ENCR_TKIP

If `WIFI_USE_WPA` and `WIFI_AES_ENABLED` are defined:

- IFPARAM_WIFI_ENCR_CCMP

IFG_WIFI_ENCRYPTION

Get encryption setting (word *).

IFS_WIFI_AUTHENTICATION

Set authentication method to use (word). Valid values are:

- IFPARAM_WIFI_AUTH_ANY
- IFPARAM_WIFI_AUTH_OPEN
- IFPARAM_WIFI_AUTH_SHAREDKEY
- IFPARAM_WIFI_AUTH_WPA_PSK
- IFPARAM_WIFI_AUTH_WPA_8021X

Note that when using WEP encryption, open authentication is actually more secure than shared key authentication.

IFG_WIFI_AUTHENTICATION

Get current authentication setting (word *).

ifconfig (cont'd)

IFS_WIFI_TX_RATE

Set maximum transmit rate, as a multiple of 100 kbps (int). Use the macros below, or their integer equivalents.

Available on 802.11b and 802.11g hardware:

- IFPARAM_WIFI_TX_RATE_ANY, 0
- IFPARAM_WIFI_TX_RATE_1, 10 (1.0 Mbps)
- IFPARAM_WIFI_TX_RATE_2, 20 (2.0 Mbps)
- IFPARAM_WIFI_TX_RATE_5_5, 55 (5.5 Mbps)
- IFPARAM_WIFI_TX_RATE_11, 110 (11.0 Mbps)

Available on 802.11g hardware only:

- IFPARAM_WIFI_TX_RATE_6, 60 (6.0 Mbps)
- IFPARAM_WIFI_TX_RATE_9, 90 (9.0 Mbps)
- IFPARAM_WIFI_TX_RATE_12, 120 (12.0 Mbps)
- IFPARAM_WIFI_TX_RATE_18, 180 (18.0 Mbps)
- IFPARAM_WIFI_TX_RATE_24, 240 (24.0 Mbps)
- IFPARAM_WIFI_TX_RATE_36, 360 (36.0 Mbps)
- IFPARAM_WIFI_TX_RATE_48, 480 (48.0 Mbps)
- IFPARAM_WIFI_TX_RATE_54, 540 (54.0 Mbps)

IFG_WIFI_TX_RATE

Get maximum transmit rate setting (int *).

IFS_WIFI_TX_POWER

Set maximum Tx power; valid values are: 0-15.

IFG_WIFI_TX_POWER

Get maximum Tx power setting (int *).

IFS_WIFI_FRAG_THRESHOLD

Set fragment threshold; valid values are: 256-2346.

IFG_WIFI_FRAG_THRESHOLD

Get fragment threshold (int *).

IFS_WIFI_RTS_THRESHOLD

Set RTS (request-to-send) threshold; valid values are: 1-2347.

IFG_WIFI_RTS_THRESHOLD

Get RTS threshold (int *).

ifconfig (cont'd)

IFS_WIFI_SCAN

Initiate a Wi-Fi scan of all valid channels for the current region. A pointer to a scan callback function is the only parameter. The callback function must have the following function prototype:

```
root void scan_callback (far wifi_scan_data *data);
```

When the scan has completed, the scan callback function is called. The Wi-Fi scan can be done without taking the interface down, but it will briefly interrupt the network connectivity as it scans the channels on the wireless network.

The scan data is provided to the callback function in its data parameter. The `wifi_scan_data` structure has the following definition:

```
typedef struct {
    int count;
    _wifi_wln_scan_bss bss[_WIFI_SCAN_NUM];
} wifi_scan_data;
```

`_WIFI_SCAN_NUM` is set to 16 in `wifi_wln_api.lib`. “count” contains the number of access points that were detected. “bss” is an array where each element corresponds to a detected access point. `_wifi_wln_scan_bss` has the following definition:

```
typedef struct {
    uint8 ssid[WLN_SSID_SIZE];           // network name, up to 32 bytes long
    int  ssid_len;                       // number of bytes in the SSID
    int  channel;                         // Wi-Fi network channel (1-13)
    mac_addr bss_addr;                   // BSS ID (the AP's MAC address)
    uint16 bss_caps;                     // reserved
    uint8 wpa_info[WLN_WPAIE_SIZE];      // reserved
    uint8 erp_info;                      // reserved
    uint16 rates;                        // reserved
    uint16 rates_basic;                  // reserved
    uint16 atim;                         // reserved
    int tx_rate;                         // max transmission rate (in 100 kbps)
    int rx_signal;                       // received signal strength
} _wifi_wln_scan_bss;
```

See the `WiFiScan.c` sample program for an example of using `IFS_WIFI_SCAN`.

ifconfig (cont'd)

IFG_WIFI_STATUS

Get current MAC status.

The `IFG_WIFI_STATUS` command returns the current Wi-Fi MAC status into the user-supplied buffer or data structure area. The buffer must be large enough to hold the entire `wln_status` structure (size can be checked with `sizeof(wln_status)`).

The `wln_status` structure has the following definition:

```
typedef struct {
    wln_state    state;                // Association state, see below.
    uint8       ssid[WLN_SSID_SIZE];  // Current service set ID (SSID)
    int         ssid_len;              // Service set ID length
    int         channel;               // Current channel: 1-13
    mac_addr    bss_addr;              // BSS ID (AP MAC address)
    uint16      bss_caps;              // reserved
    uint8       wpa_info[WLN_WPAIE_SIZE]; // reserved
    uint32      authen;                // reserved
    uint32      encrypt;               // reserved
    int         tx_rate;               // transmit rate, in 100 kbps
    int         rx_rate;               // Last received rate, in 100 kbps
    int         rx_signal;              // Last received signal strength (range 0-107)
    int         tx_power;              // reserved
    uint8       country_info[WLN_COUNTRY_STRLEN]; // reserved
    int         link;                  // reserved
} wln_status;
```

The association state is indicated by one of the following macros:

`WLN_ST_STOPPED` - Wi-Fi driver is stopped
`WLN_ST_SCANNING` - Currently performing a scan
`WLN_ST_ASSOC_ESS` - Associated with an AP
`WLN_ST_AUTH_ESS` - Authenticated with an AP
`WLN_ST_JOIN_IBSS` - Joined an existing ad-hoc network
`WLN_ST_START_IBSS` - Started an ad-hoc network

COMMANDS FOR MANAGING WEP SHARED AND WPA PRE-SHARED KEYS

IFS_WIFI_WEP_KEYNUM

Set which of four WEP keys to use (0-3).

IFG_WIFI_WEP_KEYNUM

Get WEP key number that is active (int *).

The `wln_status` structure is documented with the parameter in `ifconfig()`.

ifconfig (cont'd)

IFS_WIFI_WEP_KEY_BIN

Set WEP key to 5 or 13 bytes, using 3 parameters:

1. WEP key number to set (int, 0-3),
2. Length in bytes:
 - IFPARAM_WIFI_WEP_KEY40 (5-byte key)
 - IFPARAM_WIFI_WEP_KEY104 (13-byte key)
3. Pointer to WEP key value (byte *)

IFG_WIFI_WEP_KEY_BIN

Get value of specified WEP key, using 3 parameters:

1. WEP key number specified (int, 0-3)
2. Pointer to location to store length of WEP key (int *)
3. Pointer to location to store key value (byte *)

IFS_WIFI_WEP_KEY_HEXSTR

Set WEP key to 10 or 26 character hex string.

Set value of WEP key # specified in the first parameter (int) to 10 or 26 char hex string stored at the second parameter (char *).

IFG_WIFI_WEP_KEY_HEXSTR

Get value of WEP key specified in the first parameter (int).
Store hex string in the second parameter (char *).

IFS_WIFI_WPA_PSK_PASSPHRASE

Set the WPA PSK to hash based on a null-terminated ASCII string of up to 63 characters (char *) and the SSID. (Set the SSID before setting the passphrase.)

After generating the key, you can use IFG_WIFI_WPA_PSK_HEXSTR to get the key as a 64-character hex string for use with IFS_WIFI_WPA_PSK_HEXSTR. Note that if you change the SSID after setting the passphrase, you will need to reset the passphrase by using the IFS_WIFI_WPA_PSK_PASSPHRASE command again.

WPA pre-shared keys (PSK) are used for WPA PSK authentication and for TKIP and CCMP encryption. Setting the key with a passphrase can take 20 seconds on an RCM4400W. Setting the key directly as hex is much more efficient.

IFS_WIFI_WPA_PSK_HEXSTR

Set WPA PSK to 64-character hex string (char[65]).

IFS_WIFI_WPA_PSK_HEXSTR takes a null-terminated ASCII string of 64 hex digits and uses it for the key.

IFG_WIFI_WPA_PSK_HEXSTR

Get WPA PSK as a 64-character hex string (char[65]).

ifconfig (cont'd)

IFS_WIFI_WPA_PSK_BIN

Set 32-byte WPA PSK (byte[32]).

IFG_WIFI_WPA_PSK_BIN

Get 32-byte WPA PSK (byte[32]).

COMMANDS FOR WI-FI ROAMING:**IFS_WIFI_ROAM_ENABLE**

Set roaming state: on/off (bool).

IFG_WIFI_ROAM_ENABLE

Get roaming enabled state (bool *).

IFS_WIFI_ROAM_BEACON_MISS

Set number of beacons that must be missed consecutively before a scan for a new access point is initiated (int).

IFG_WIFI_ROAM_BEACON_MISS

Get number of beacons that must be missed consecutively before a scan for a new access point is initiated (int *).

COMMANDS FOR WPA/WPA2 IN ENTERPRISE MODE**IFS_WIFI_WPA_PROTOCOL****IFG_WIFI_WPA_PROTOCOL**

Set/Get acceptable security protocol(s) [word/word *]. Default is to support both WPA and WPA2.

- IFPARAM_WIFI_WPA_PROTOCOL_WPA - WPA; aka, IEEE 802.11i/D3.0
- IFPARAM_WIFI_WPA_PROTOCOL_WPA2 - WPA2; aka, IEEE 802.11i/D9.0
- IFPARAM_WIFI_WPA_PROTOCOL_RSN - WPA2; aka, IEEE 802.11i/D9.0
- IFPARAM_WIFI_WPA_PROTOCOL_ALL - A bitwise combination of all protocols

IFS_WIFI_PAIRWISE_ENCRYPTION**IFG_WIFI_PAIRWISE_ENCRYPTION**

Set/Get Wi-Fi pairwise encryption types [word/word *I].

Valid values are:

- IFPARAM_WIFI_ENCR_TKIP - Valid if WIFI_USE_WPA is defined.
- IFPARAM_WIFI_ENCR_CCMP - Valid if both WIFI_USE_WPA and WIFI_AES_ENABLED are defined.
- IFPARAM_WIFI_ENCR_NONE - Use only Group Keys (deprecated, should not be included if APs support pairwise keys).

ifconfig (cont'd)

IFS_WIFI_EAP_METHODS

IFG_WIFI_EAP_METHODS

Set/Get acceptable EAP method [longword/longword *].

The parameter is a bitmask with one of the following values. Note that methods that do not have compiled-in support via the WPA_USE_EAP macro will be ignored. Currently, we only support EAP_TYPE_TLS and EAP_TYPE_PEAP.

Valid values are:

- IFPARAM_EAP_PEAP - This parameter selects PEAP-MSCHAPV2. Using this method, requires the statement:
#define WPA_USE_EAP WPA_USE_EAP_PEAP
- IFPARAM_EAP_TLS - This parameter selects EAP-TLS. Using this method, requires the statement:
#define WPA_USE_EAP WPA_USE_EAP_TLS

IFS_WIFI_IDENTITY

IFG_WIFI_IDENTITY

Set/Get identity string for EAP [char */char **].

IFS_WIFI_ANONYMOUS_IDENTITY

IFG_WIFI_ANONYMOUS_IDENTITY

Set/Get anonymous identity string for “outer” EAP [char */char **].

IFS_WIFI_PASSWORD

IFG_WIFI_PASSWORD

Set/Get password string for EAP [char */char **].

IFS_WIFI_EAP_PSK_PASSPHRASE

Set Wi-Fi EAP PSK using passphrase [char *].

IFS_WIFI_EAP_PSK_HEXSTR

IFG_WIFI_EAP_PSK_HEXSTR

Set/Get Wi-Fi EAP PSK using 64-char hex string [char */char *].

IFS_WIFI_EAP_PSK_BIN

IFG_WIFI_EAP_PSK_BIN

Set/Get Wi-Fi EAP PSK using 32-byte array [byte */byte *].

ifconfig (cont'd)

IFS_WIFI_SUBJECT_MATCH

Set substring to be matched against the subject of the authentication server certificate [char *].

The subject string is in following format (for example):

C=US/ST=CA/L=Davis/CN=Test1-AS/emailAddress=test1_as@rabbit.com

so this parameter string could be set to “/L=Davis/CN=Test” to allow all access servers with a location of “Davis” and a common name starting with “Test.”

IFS_WIFI_ALTSUBJECT_MATCH

Set semicolon separated string of entries to be matched against the alternative subject name of the authentication server certificate [char *].

If this string is set, the server certificate is only accepted if it contains one of the entries in an alternative subject name extension.

altSubjectName string is in following format: TYPE:VALUE

Example: EMAIL:server@example.com

Example: DNS:server.example.com;DNS:server2.example.com

Following types are supported: EMAIL, DNS, URI.

COMMANDS FOR CERTIFICATES AND KEYS

The following parameters are for convenience when “hard-coding” certificates (e.g., from #ximported files).

Pass a zero longword as the parameter in order to delete any resources that were allocated on a previous call (since the library manages the SSL_Cert_t structures that are created).

IFS_WIFI_CA_CERT

IFG_WIFI_CA_CERT

Set/Get CA certificate [SSL_Cert_t far */SSL_Cert_t far **].

The authentication server is not verified if no CA certificate is provided in the “Set” command, which introduces risk of “access server spoofing.”

IFS_WIFI_CA_CERT_PATH

Set CA certificate as a Zserver resource path [char *].

ifconfig (cont'd)

IFS_WIFI_CLIENT_CERT

IFG_WIFI_CLIENT_CERT

Set/Get client certificate [SSL_Cert_t far */SSL_Cert_t far **].

IFS_WIFI_CLIENT_CERT_PATH

Set client certificate as Zserver file [char *,char *].

The first parameter is for the certificate, and the second one is for the private key. If the certificate parameter is NULL, then delete resources.

IFS_WIFI_CA_CERT_XIM

Set CA certificate as #ximport DER/PEM format [longword].

IFS_WIFI_CLIENT_CERT_XIM

Set client certificate and private key as two #ximport DER/PEM format files [longword, longword]. The first parameter is the certificate, the second one is for the private key.

RETURN VALUE

0: Success.

>0: identifier of first parameter group that encountered an error.

-1: iface parameter is invalid.

An exception (runtime error) is raised if the parameter list contains an invalid parameter number.

LIBRARY

NET.LIB

SEE ALSO

`sock_init`, `tcp_config`, `ip_print_ifs`, `ifstatus`, `ifpending`

ifdown

```
int ifdown( int iface );
```

DESCRIPTION

This function attempts to deactivate the specified interface. The action depends on the previous status reported by `ifpending()`.

<code>ifpending()</code>	Action
<code>IF_DOWN</code>	None, returns <code>IFCTL_OK</code> .
<code>IF_COMING_UP</code>	Cancels bringing up, starts down.
<code>IF_UP</code>	Starts bringing interface down using normal “graceful” procedure.
<code>IF_COMING_DOWN</code>	Forces down, cuts short any normal termination procedure.

PARAMETER

iface Interface number. Use one of the definitions:

- `IF_DEFAULT`
- `IF_ETH0`
- `IF_ETH1`
- `IF_WIFI0`
- `IF_PPPOE0`
- `IF_PPPOE1`
- `IF_PPP0-5`

If the interface does not exist you will get a compile time error.

`IF_DEFAULT` refers to the default interface, which will be equivalent to the first of the other interface identifiers in the above list which exists.

RETURN VALUE

`IFCTL_OK`: if OK

`IFCTL_FAIL`: if error

`IFCTL_PEND`: if OK but not complete

LIBRARY

`NET.LIB`

SEE ALSO

`ifconfig`, `ifup`, `ifstatus`, `ifpending`

ifpending

```
int ifpending( int iface );
```

DESCRIPTION

Returns indication of whether the specified interface is up, down, pending up or pending down. This gives more than `ifstatus()`, which only indicates the current state (up or down).

NOTE: ANDing the return value with 0x01 indicates a pending condition; ANDing with 0x02 is equivalent to the return from `ifstatus()`, except that `ifstatus()` returns “1” instead of “2”.

Example:

```
// assume IF_DEFAULT is down. Bring it up...
ifconfig(IF_DEFAULT, IFS_UP, IFS_END);
while (ifpending(IF_DEFAULT) == IF_COMING_UP)
tcp_tick(NULL);
if (ifpending(IF_DEFAULT) == IF_UP) // came up OK
else // failed to come up
```

PARAMETERS

iface	Interface number. Use one of the definitions: <ul style="list-style-type: none">• IF_DEFAULT• IF_ETH0• IF_ETH1• IF_WIFI0• IF_PPPOE0• IF_PPPOE1• IF_PPP0-5
--------------	---

If the interface does not exist, you will get a compile time error.

IF_DEFAULT refers to the default interface, which will be equivalent to the first of the other interface identifiers in the above list which exists.

RETURN VALUE

- 0: If interface is currently down and not pending up.
- 1: If interface is currently down and pending up.
- 2: If interface is currently up and not pending down.
- 3: If interface is currently up and pending down.

Rather than the above constants, you should use the macros: `IF_DOWN`, `IF_COMING_UP`, `IF_UP` and `IF_COMING_DOWN`.

LIBRARY

NET.LIB

SEE ALSO

[ifconfig](#), [ifdown](#), [ifup](#), [ifstatus](#)

ifstatus

```
int ifstatus( int iface );
```

DESCRIPTION

This macro returns the status of the specified interface.

PARAMETER

iface Interface number. Use one of the definitions

- IF_DEFAULT
- IF_ETH0
- IF_ETH1
- IF_WIFI0
- IF_PPPOE0
- IF_PPPOE1
- IF_PPP0-5

If the interface does not exist, then you will get a compile time error.
IF_DEFAULT refers to the default interface, which will be equivalent to the first of the other interface identifiers in the above list which exists.

RETURN VALUE

0: Interface is currently down.
!0: Interface is currently up (active).

LIBRARY

NET.LIB

SEE ALSO

[ifconfig](#), [ifup](#), [ifdown](#), [ifpending](#)

ifup

```
int ifup( int iface );
```

DESCRIPTION

This function attempts to activate the specified interface. The action depends on the previous status reported by `ifpending()`.

Return Value of <code>ifpending()</code>	Action Taken by <code>ifup()</code>
<code>IF_UP</code>	None, returns <code>IFCTL_OK</code>
<code>IF_COMING_UP</code>	None, returns <code>IFCTL_PEND</code>
<code>IF_DOWN</code>	Starts bringing interface up using normal procedure.
<code>IF_COMING_DOWN</code>	<p>Depends on interface capabilities, but note that in practice this always returns <code>IFCTL_FAIL</code>.</p> <p>If the interface is currently “coming down,” the action taken when <code>ifup()</code> is called depends on the interface. Some interfaces may not allow the interface to come back up in this condition, in which case <code>IFCTL_FAIL</code> will be returned. Try again later when the interface is fully down, or you can call <code>ifdown()</code> to force the interface down immediately.</p> <p>If the interface allows changing requested state when coming down, then the return value will be either <code>IFCTL_PEND</code> or <code>IFCTL_OK</code>. Currently, the only interface types which do not come down instantly are PPP over async serial and PPPoE. These do NOT currently implement calling <code>ifup()</code> when they are pending down.</p>

ifup (cont'd)

PARAMETER

iface Interface number. Use one of the definitions

- IF_DEFAULT
- IF_ETH0
- IF_ETH1
- IF_WIFI0
- IF_PPPOE0
- IF_PPPOE1
- IF_PPP0-5

If the interface does not exist, then you will get a compile time error.
IF_DEFAULT refers to the default interface, which will be equivalent to the first of the other interface identifiers in the above list which exists.

RETURN VALUE

IFCTL_OK: if OK.
IFCTL_FAIL: if error.
IFCTL_PEND: if OK but not complete.

LIBRARY

NET.LIB

SEE ALSO

[ifconfig](#), [ifdown](#), [ifstatus](#)

`inet_addr`

```
longword inet_addr( char *dotted_ip_string );
```

DESCRIPTION

Converts an IP address from dotted decimal IP format to its binary representation. No check is made as to the validity of the address.

PARAMETERS

`dotted_ip_string` Dotted decimal IP string, e.g., "10.10.6.100".

RETURN VALUE

0: Failure.

Binary representation of `dotted_ip_string`: Success.

LIBRARY

IP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[inet_ntoa](#)

inet_ntoa

```
char *inet_ntoa( char *s, longword ip );
```

DESCRIPTION

Converts a binary IP address to its dotted decimal format, e.g.,
`inet_ntoa(s, 0x0a0a0664)` returns a pointer to "10.10.6.100".

PARAMETERS

s	Location to place the dotted decimal string. A sufficient buffer size would be 16 bytes.
ip	The IP address to convert.

RETURN VALUE

Pointer to the dotted decimal string pointed to by `s`.

LIBRARY

IP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[inet_addr](#)

ip_iface

```
byte ip_iface( longword ipaddr, int local_only );
```

DESCRIPTION

Given an IP address, this function return the interface number for that address. If `ipaddr` is an address on one of the local subnets, then the interface to that subnet is returned.

If the address is not local, then the `local_only` parameter determines the result:

If `local_only` is 1, then `IF_ANY` will be returned for a non-local address.

Otherwise, the `router_for()` function is invoked to find the correct router -- the interface for the router is returned.

PARAMETERS

ipaddr	IP address of an external host.
local_only	0: allow non-local addresses (returns interface for router). 1: return <code>IF_ANY</code> for non-local addresses.

RETURN VALUE

Interface number (0..`IF_MAX-1`), of possibly `IF_ANY` (0xFF).

LIBRARY

`IP.LIB`

SEE ALSO

[router_for](#)

ip_print_ifs

```
void ip_print_ifs( void );
```

DESCRIPTION

Print all interface table entries. This is for debugging only, since the results are printed to the Dynamic C Stdio window.

There are 8 fields for each interface entry:

#	The interface number
IP addr	The local ("home") IP address of this interface. May be 0.0.0.0 if interface is not currently active.
Mask	Local subnet mask.
Up	Indicates status; one of Yes: interface currently active No: currently inactive NYU: Not Yet Up i.e., coming up NYD: Not Yet Down i.e., coming down
Type:	Interface type; one of eth: normal Ethernet ppp: PPP over serial port pppoe: PPP over Ethernet
MTU:	Maximum transmission unit.
Flags:	A list of the following characters: *: this is the default interface (IF_DEFAULT) D: Use DHCP DD: Currently configured via DHCP S: allow IP addr configuration via directed ping (ICMP echo). SS: IP address currently set via directed ping 1: IGMP version 1 router present on this interface
Peer/router	IP address of peer node (for PPP or PPPoE), or address of default router on this interface (for Ethernet type). May be blank or 0.0.0.0 if no peer or router is available.

LIBRARY

IP.LIB

`ip_timer_expired`

```
word ip_timer_expired( void *s );
```

DESCRIPTION

Check the timer inside the socket structure that was set by `ip_timer_init()`.

PARAMETER

s Pointer to a socket.

RETURN VALUE

0: If not expired.

1: If expired.

LIBRARY

NET.LIB

SEE ALSO

[`ip_timer_init`](#)

`ip_timer_init`

```
void ip_timer_init( void *s, word seconds );
```

DESCRIPTION

Set a timer inside the socket structure.

PARAMETER

s	Pointer to a socket.
seconds	Number of seconds for the time-out; if <code>seconds</code> is zero never time-out.

RETURN VALUE

None.

LIBRARY

NET.LIB

SEE ALSO

[ip_timer_expired](#)

`is_valid_iface`

```
int is_valid_iface( int iface );
```

DESCRIPTION

This function returns a boolean indicator of whether the given interface number is valid for the configuration.

PARAMETER

iface	Interface number. Use one of the definitions <ul style="list-style-type: none">• <code>IF_ETH0</code>• <code>IF_ETH1</code>• <code>IF_WIFI0</code>• <code>IF_PPPOE0</code>• <code>IF_PPPOE1</code>• <code>IF_PPP0-5</code>
--------------	---

RETURN VALUE

!0: Interface is valid.
0: Interface does not exist.

LIBRARY

`NET.LIB`

SEE ALSO

`ifconfig`, `ifup`, `ifdown`, `ifstatus`

ModemClose

```
void ModemClose( void );
```

DESCRIPTION

Closes the serial driver down.

LIBRARY

MODEM.LIB

ModemConnected

```
int ModemConnected( void );
```

DESCRIPTION

Returns true if the DCD line is asserted, meaning the modem is connected to a remote carrier.

RETURN VALUE

1: DCD line is active.

0: DCD inactive (nothing connected).

LIBRARY

MODEM.LIB

ModemExpect

```
int ModemExpect( char *send_string, unsigned long timeout );
```

DESCRIPTION

Listens for a specific string to be sent by the modem.

PARAMETERS

send_string A NULL-terminated string to listen for.

timeout Maximum wait in milliseconds for a character.

RETURN VALUE

1: The expected string was received.

0: A timeout occurred before receiving the string.

LIBRARY

MODEM.LIB

ModemHangup

```
int ModemHangup( void );
```

DESCRIPTION

Sends "ATH" and "ATZ" commands.

RETURN VALUE

1: Success.

0: Modem not responding.

LIBRARY

MODEM.LIB

ModemInit

```
int ModemInit( void );
```

DESCRIPTION

Resets modem with AT, ATZ commands.

RETURN VALUE

1: Success.

0: Modem not responding.

LIBRARY

MODEM.LIB

ModemOpen

```
int ModemOpen( unsigned long baud );
```

DESCRIPTION

Starts up communication with an external modem.

PARAMETERS

baud The baud rate for communicating with the modem.

RETURN VALUE

1: External modem detected

0: Not connected to external modem

LIBRARY

MODEM.LIB

ModemReady

```
int ModemReady( void );
```

DESCRIPTION

Returns true if the DSR line is asserted.

RETURN VALUE

1: DSR line is active.

0: DSR inactive (nothing connected).

LIBRARY

MODEM.LIB

ModemRinging

```
int ModemRinging( void );
```

DESCRIPTION

Returns true if the RI line is asserted, meaning that the line is ringing.

RETURN VALUE

1: RI line is active.

0: RI inactive (nothing connected).

LIBRARY

MODEM.LIB

ModemSend

```
void ModemSend( char *send_string );
```

DESCRIPTION

Sends a string to the modem.

PARAMETERS

send_string A NULL-terminated string to be sent to the modem.

LIBRARY

MODEM.LIB

ModemStartPPP

```
void ModemStartPPP( void );
```

DESCRIPTION

Hands control of the serial line over to the PPP driver.

LIBRARY

MODEM.LIB

multicast_joingroup

```
int multicast_joingroup( int iface, longword ipaddr );
```

DESCRIPTION

This function joins the specified multicast group (class D IP address--from 224.0.0.0 to 239.255.255.255) on the specified interface. For an Ethernet interface, it configures the hardware to accept multicast packets for the specified address.

Note that this function is called automatically when `udp_open()` is used to open a multicast address.

PARAMETER

iface	Interface on which to join the group. Use one of the definitions <ul style="list-style-type: none">• IF_ETH0• IF_ETH1• IF_WIFI0• IF_DEFAULT
ipaddr	Multicast group to join.

RETURN VALUE

0: Success.

1: Failure (e.g., `ipaddr` is not a multicast address; or not enough available ARP entries to hold the group).

LIBRARY

IGMP.LIB

multicast_leavegroup

```
int multicast_leavegroup( int iface, longword ipaddr );
```

DESCRIPTION

This function leaves the specified multicast group (class D IP address--from 224.0.0.0 to 239.255.255.255) on the specified interface. For an Ethernet interface, it configures the hardware to no longer accept multicast packets for the specified address. This function will leave the group no matter how many `multicast_joingroup()` calls were made on that group. However, note that this function will not actually leave a group for which there are UDP sockets. However, when those UDP sockets close, the group will be left.

Note that this function is called automatically when a multicast UDP socket is closed.

PARAMETER

iface	Interface on which to leave the group. Use one of the definitions <ul style="list-style-type: none">• IF_ETH0• IF_ETH1• IF_WIFI0• IF_DEFAULT
ipaddr	Multicast group to leave.

RETURN VALUE

0: Success.
1: Failure (e.g., `ipaddr` is not a multicast address).

LIBRARY

IGMP.LIB

ntohl

```
longword ntohl( longword value );
```

DESCRIPTION

Converts network-ordered long word to host-ordered long word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order.

PARAMETERS

value Network-ordered long word.

RETURN VALUE

Network-ordered long word in host-ordered format, e.g., `ntohl(0x44332211)` returns `0x11223344`

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[htons](#), [ntohs](#), [htonl](#)

ntohs

```
word ntohs( word value );
```

DESCRIPTION

Converts network-ordered word to host-ordered word. This function is necessary if you are implementing standard internet protocols because the Rabbit does not use the standard for network byte ordering. The network orders bytes with the most significant byte first and the least significant byte last. On the Rabbit, the bytes are in the opposite order.

PARAMETERS

value Network-ordered word.

RETURN VALUE

Network-ordered word in host-ordered format, e.g., `ntohs(0x2211)` returns `0x1122`

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[htonl](#), [ntohl](#), [htons](#)

pd_getaddress

```
void pd_getaddress( int nic, void *buffer );
```

DESCRIPTION

This function copies the Ethernet address (aka the MAC address) into the buffer.

PARAMETERS

nic	Starting with Dynamic C 7.30, this parameter identifies an Ethernet interface. Use a value of 0 if only one NIC is present
buffer	Place to copy address to. Must be at least 6 bytes.

RETURN VALUE

None.

LIBRARY

PKTDRV.LIB

EXAMPLE

```
main() {
    char buf[6];
    sock_init();
    pd_getaddress(0,buf);
    printf("Your Link Address is:%02x%02x:%02x%02x:%02x%02x
        \n", buf[0], buf[1], buf[2], buf[3], buf[4], buf[5]);
}
```

pd_havelink

```
int pd_havelink( int nic );
```

DESCRIPTION

Determines if the physical-layer link is established for the specified NIC.

PARAMETERS

nic The NIC to check. Use a value of 0 if only one NIC is present.

RETURN VALUE

0: There is no link.

!0: The link is established.

LIBRARY

PKTDRV.LIB

pd_powerdown

```
int pd_powerdown( int nic );
```

DESCRIPTION

Power down the interface hardware by turning off as many services as possible. When the NIC is in powerdown mode, it is very important to *not* call any TCP/IP, Ethernet, etc. functions, as they will obviously fail, and the results will be undefined. `pd_powerup()` should be the very next network function called, to re-enable the NIC.

WARNING: When used to power down a wireless chipset (IF_WIFIO interface) then this will result in loss of association with the access point. In effect, the network interface will come down. Thus, it is highly recommended to bring the interface down before calling this power-down function. See the `POWERDOWN.C` sample in `Samples\RCM4400W\TCP/IP` for an example of how to do this.

PARAMETERS

nic	The interface to powerdown. Currently, only some Ethernet chipsets are applicable (IF_ETH0), and some Wi-Fi chipsets such as on the RCM4400W. If the interface does not support this function, then it will be ignored.
------------	---

RETURN VALUE

0: Success.
!0: Error.

LIBRARY

PKTDRV.LIB

SEE ALSO

[pd_powerup](#)

pd_powerup

```
int pd_powerup( int nic );
```

DESCRIPTION

Power up the NIC, undoing the sleepy-mode changes made by `pd_powerdown`. After this function has returned success, Ethernet and TCP/IP function may be called again.

NOTE: This function will block for 10 ms, to let the chip start up.

PARAMETERS

nic The NIC to power up. Use a value of 0 if only one NIC is present.

RETURN VALUE

0: Success.

!0: Error.

LIBRARY

PKTDRV.LIB

SEE ALSO

[pd_powerdown](#)

`_ping`

```
int _ping( longword host_ip, longword sequence_number );
```

DESCRIPTION

Generates an ICMP request for host. NOTE: this is a macro that calls `_send_ping`.

PARAMETERS

<code>host_ip</code>	IP address to send ping.
<code>sequence_number</code>	User-defined sequence number.

RETURN VALUE

0: Success.
1: Failure, unable to resolve hardware address.
-1: Failure, unable to transmit ICMP request.

LIBRARY

`ICMP.LIB`

SEE ALSO

`_chk_ping`, `_send_ping`

PPPactive

```
int PPPactive( void );
```

DESCRIPTION

Returns boolean value indicating if there is currently an active link to a peer.

RETURN VALUE

>0: Active link to peer.

0: No active link.

LIBRARY

PPP.LIB

PPPnegotiateIP

```
void PPPnegotiateIP( unsigned long local_ip, unsigned long remote_ip );
```

DESCRIPTION

Sets PPP driver to negotiate IP addresses for itself and the remote peer. Otherwise, the system will rely on the remote peer to set addresses.

PARAMETERS

local_ip IP number to use for this PPP connection.

remote_ip IP number that the remote peer should be set to.

RETURN VALUE

None.

LIBRARY

PPP.LIB

PPPsetAuthenticatee

```
void PPPsetAuthenticatee( char * username, char * password );
```

DESCRIPTION

Sets up the driver to send a PAP authentication message to a peer when requested.

PARAMETERS

username	The username to send to the peer. The argument string is not copied, so it must stay constant.
password	The password to send to the peer. The argument string is not copied, so it must stay constant.

RETURN VALUE

None.

LIBRARY

PPP.LIB

PPPsetAuthenticator

```
void PPPsetAuthenticator( char * username, char * password );
```

DESCRIPTION

Sets up the driver to require a PAP authentication message from a peer. Negotiation will fail unless the peer sends the specified username/password pair. This function is generally used when the Rabbit is acting as a dial-in server.

PARAMETERS

username	The user name that the peer must match for the link to proceed. The argument string is not copied, so it must stay constant.
password	The password that the peer must match for the link to proceed. The argument string is not copied, so it must stay constant.

RETURN VALUE

None.

LIBRARY

PPP.LIB

PPPshutdown

```
int PPPshutdown( unsigned long timeout );
```

DESCRIPTION

Sends a Link Terminate Request packet. Waits for link to be torn down.

PARAMETERS

timeout Number of milliseconds to wait before giving up on a response from the peer.

RETURN VALUE

1: Shutdown succeeded.
0: Shutdown timed-out.

LIBRARY

PPP.LIB

psocket

```
void psocket( void *s );
```

DESCRIPTION

Given an open UDP or TCP socket, the IP address of the remote host is printed out to the Stdio window in dotted IP format followed by a colon and the decimal port number on that machine. This routine can be useful for debugging your programs.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

None.

LIBRARY

BSDNAME.LIB

resolve

```
longword resolve( char *host_string );
```

DESCRIPTION

Converts a text string, which contains either the dotted IP address or host name, into the longword containing the IP address. In the case of dotted IP, no validity check is made for the address. NOTE: this function blocks. Names are currently limited to 64 characters. If it is necessary to lookup larger names include the following line in the application program:

```
#define DNS_MAX_NAME <len in chars>
```

If `DISABLE_DNS` has been defined, `resolve()` will not do DNS lookup.

If you are trying to resolve a host name, you must set up at least one name server. You can set the default name server by defining the `MY_NAMESERVER` macro at the top of your program. When you call `resolve()`, it will contact the name server and request the IP address. If there is an error, `resolve()` will return 0L.

To simply convert dotted IP to longword, see `inet_addr()`.

For a sample program, see the Example Using `tcp_open()` listed under `tcp_open()`.

PARAMETERS

host_string Pointer to text string to convert.

RETURN VALUE

0: Failure.

!0: The IP address `*host_string` resolves to.

LIBRARY

DNS.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[_arp_resolve](#), [inet_addr](#), [inet_ntoa](#)

resolve_cancel

```
int resolve_cancel( int handle );
```

DESCRIPTION

Cancels the resolve request represented by the given handle. If the handle is 0, then this function cancels all outstanding resolve requests.

PARAMETERS

handle Handle that represents a DNS lookup process, or 0 to cancel all outstanding resolve requests.

RETURN VALUE

RESOLVE_SUCCESS: The resolve request has been cancelled and is no longer valid.

RESOLVE_HANDLENOTVALID: There is no request for the given handle.

RESOLVE_NONAMESERVER: No nameserver has been defined.

LIBRARY

DNS.LIB

SEE ALSO

[resolve_name_start](#), [resolve_name_check](#), [resolve](#)

`resolve_name_check`

```
int resolve_name_check( int handle, longword *resolved_ip );
```

DESCRIPTION

Checks if the DNS lookup represented by the given handle has completed. On success, it fills in the resolved IP address in the space pointed to by `resolved_ip`.

PARAMETERS

handle	Handle that represents a DNS lookup process.
resolved_ip	A pointer to a user-supplied longword where the resolved IP address will be placed.

RETURN VALUE

RESOLVE_SUCCESS: The address was resolved. The given handle will no longer be valid after this value is returned.

RESOLVE_AGAIN: The resolve process has not completed, call this function again.

RESOLVE_FAILED: The DNS server responded that the given host name does not exist. The given handle will no longer be valid if **RESOLVE_FAILED** is returned.

RESOLVE_TIMEDOUT: The request has been cancelled because a response from the DNS server was not received before the last time-out expired. The given handle will no longer be valid after this value is returned.

RESOLVE_HANDLENOTVALID: There is no DNS lookup occurring for the given handle.

RESOLVE_NONAMESERVER: No nameserver has been defined.

LIBRARY

`DNS.LIB`

SEE ALSO

[resolve_name_start](#), [resolve_cancel](#), [resolve](#)

resolve_name_start

```
int resolve_name_start( char *hostname );
```

DESCRIPTION

Starts the process of resolving a host name into an IP address. The given host name is limited to `DNS_MAX_NAME` characters, which is 64 by default (63 characters + the `NULL` terminator). If a default domain is to be added, then the two strings together are limited to `DNS_MAX_NAME`.

If `hostname` does not contain a `'.'` then the default domain (`MY_DOMAIN`), if provided, is appended to `hostname`. If `hostname` with the appended default domain does not exist, `hostname` is tried by itself. If that also fails, the lookup fails.

If `hostname` does contain a `'.'` then `hostname` is looked up by itself. If it does not exist, the default domain is appended, and that combination is tried. If that also fails, the lookup fails.

If `hostname` ends with a `'.'`, then the default domain is not appended. The host name is considered “fully qualified.” The lookup is attempted without the ending `'.'` and if that fails no other combinations are attempted.

This function returns a handle that must be used in the subsequent `resolve_name_check()` and `resolve_cancel()` functions.

PARAMETERS

hostname Host name to convert to an IP address

RETURN VALUE

>0: Handle for calls to `resolve_name_check()` and `resolve_cancel()`.

`RESOLVE_NOENTRIES`: Could not start the resolve process because there were no resolve entries free.

`RESOLVE_LONGHOSTNAME`: The given hostname was too large.

`RESOLVE_NONAMESERVER`: No nameserver has been defined.

LIBRARY

`DNS.LIB`

SEE ALSO

`resolve_name_check`, `resolve_cancel`, `resolve`

rip

```
char *rip( char *string );
```

DESCRIPTION

Strips newline (\n) and/or carriage return (\r) from a string. Only the first \n and \r characters are replaced with \0s. The resulting string beyond the first \0 character is undefined.

PARAMETERS

string Pointer to a string.

RETURN VALUE

Pointer to the modified string.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

EXAMPLE

```
setmode( s, TCP_MODE_ASCII );  
...  
sock_puts( s, rip( questionable_string ) );
```

NOTE: In ASCII mode `sock_puts()` adds \n; `rip` is used to make certain the string does not already have a newline character. Remember, `rip` modifies the source string, not a copy!

router_add

```
ATHandle router_add( longword ipaddr, byte iface, longword subnet,
                    longword mask, word flags );
```

DESCRIPTION

Add a router to the router table. The same router can be added multiple times, with different subnet and mask. Normally, only one entry is needed in order to access non-local subnets: this entry should be specified with a zero mask. The hardware address of the router is not immediately resolved, however this can be done explicitly by calling `arpresolve_start()` with the same IP address. Otherwise, the router will be resolved only when it first becomes necessary.

PARAMETERS

ipaddr	IP address of the router. This address should be on the local subnet, since non-local routers are not supported.
iface	Interface to use to access this router, or <code>IF_DEFAULT</code> .
subnet	Subnet accessible through this entry.
mask	Subnet mask for this entry.
flags	Flags word: set to zero (non-zero reserved for internal use).

RETURN VALUE

Positive value: completed successfully. The return value is the ARP cache table entry for this router.

`ATH_NOENTRIES`: insufficient space in either the router or ARP cache tables.

LIBRARY

`ARP.LIB`

router_del_all

```
void router_del_all( void );
```

DESCRIPTION

Delete all router table entries. This will make any host that is not on the local subnet inaccessible. This function is usually called in preparation for adding a new router entry.

LIBRARY

ARP.LIB

router_delete

```
ATHandle router_delete( longword ipaddr );
```

DESCRIPTION

Delete a router from the router table. All instances of the router's IP address are deleted, and the ARP cache table entry is flushed.

PARAMETER

ipaddr IP address of the router. This address should be on the local subnet, since non-local routers are not supported.

RETURN VALUE

Positive value: completed successfully.

ATH_NOTFOUND: specified entry did not exist.

LIBRARY

ARP.LIB

router_for

```
ATHandle router_for( longword ipaddr, byte *router_used,  
    byte *r_iface );
```

DESCRIPTION

Return the ARP cache table entry corresponding to the router that handles the given IP address. If there is a pre configured router for the given address, it is selected. Otherwise, routers discovered via DHCP or ICMP router discovery are searched, with the highest preference being selected. Failing this, if there is a point-to-point interface, this is selected as the default.

An alternative mode of calling this function is invoked if `ipaddr` is zero. In this case, the default router for the specified interface (`*r_iface`) is returned. If `r_iface` is NULL, then the default interface is assumed: `IF_DEFAULT`, the only interface supported at present.

`IF_DEFAULT` may refer to the primary Ethernet NIC or a PPP connection that uses a serial port or the primary Ethernet NIC.

PARAMETERS

ipaddr	IP address of the host which is not on the local subnet.
router_used	If not NULL, the byte at this location is set to the index of the router in the router table.
r_iface	If not NULL, the byte at this location is set to the interface number that can access the router.

RETURN VALUE

Positive value: completed successfully.

`ATH_NOROUTER`: no suitable router found. Either no router is configured, or the given IP address is on the local subnet.

LIBRARY

`ARP.LIB`

router_for_iface

```
ATHandle router_for_iface( longword ipaddr, word * router_used,  
    word * r_iface, word siface);
```

DESCRIPTION

Return the ARP cache table entry corresponding to the router which handles the given IP address on a specific interface.

If the “siface” parameter is `IF_ANY`, then return the ARP cache table entry corresponding to the router which handles the given IP address. If there is a preconfigured router for the given address, it is selected. Otherwise, routers discovered via DHCP or ICMP router discovery are searched, with the highest preference being selected. Failing this, if there is a point-to-point interface, this is selected as the default.

If the specified IP address already has an entry in the ARP cache table, and there is a router already defined for that address, then the current router is returned. This overrides any other selection process.

Where multiple routers can be used to access the given IP address, the one whose entry has the most specific netmask is used. If the netmasks are equal, then an arbitrary entry is used.

An alternative mode of calling this function is invoked if “ipaddr” is zero. In this case, the default router for the specified interface (i.e., “siface”) is returned. If “siface” is `IF_ANY`, then the most general router entry is returned, i.e., the overall default.

If “siface” is a virtual ethernet interface, then if there is no specific router defined, then the router for the underlying real interface will be used. This may not be satisfactory if the router is not on the subnet defined for the virtual interface. Thus, if non-overlapping subnets are used then the virtual interface should have a valid router explicitly defined.

PARAMETERS

ipaddr	IP address of the host which is not on the local subnet, or zero to find a default router.
router_used	If not NULL, the word at this location is set to the index of the router in the router table.
r_iface	If ipaddr is non-zero this is an output parameter: If not NULL, the word at this location is set to the interface number that can access the router.
siface	Specific interface, or <code>IF_ANY</code> to select the best router for the given IP address.

router_for_iface (cont'd)

RETURN VALUE

Positive value: Completed successfully.

ATH_NOROUTER: No suitable router found. Either no router is configured, or the given IP address is on the local subnet of the specified interface, or no router on the specified interface can handle the address based on the router network/netmask information in the router table.

LIBRARY

ARP.LIB

SEE ALSO

[router_add](#)

router_print

```
int router_print( byte r );
```

DESCRIPTION

Print a router table entry, indexed by 'r.' This is for debugging only, since the results are printed to the Dynamic C stdio window. 'r' may be obtained from the `router_for()` function, by passing `&r` as the `router_used` parameter to that function.

If the specified router entry is not in use, nothing is printed and the return value is non-zero. Otherwise, the information is printed and zero returned.

See `router_printall()` for a description of the output fields printed.

PARAMETER

<code>r</code>	Router table index. A number from 0 through <code>(ARP_ROUTER_TABLE_SIZE-1)</code> .
----------------	--

RETURN VALUE

0: Success, information printed to stdio window.

!0: Entry is not in use.

LIBRARY

`ARP.LIB`

SEE ALSO

`router_printall`

router_printall

```
int router_printall( void );
```

DESCRIPTION

Print all router table entries. This is for debugging only, since the results are printed to the Dynamic C stdio window. If no routers exist in the table, nothing is printed and the return value is non-zero.

There are 6 fields for each router entry:

Router Table Entry Field	Description of Field
#	The entry number.
Flags	A list of the following characters: P = this entry pre configured T = transient entry D = added by DHCP/BOOTP R = added by ICMP redirect ? = router not reachable H = router's hardware address resolved
Address	Either the router's IP address or an indication that the entry is a point-to-point link.
i/f	Interface number.
Net/preference	For pre configured entries, indicates the network(s) which are served by this entry (the Mask indicates which bits of the IP address are used to match with the network address). For non-pre configured entries, this is the "preference value" assigned.
Mask/exp(sec)	For pre configured entries, the bitmask to apply to IP addresses when matching against the above network. Otherwise, is the expiry time from the present, in seconds, of a transient entry.

RETURN VALUE

0: Success, information printed to stdio window.

!0: No routers in the table.

LIBRARY

ARP.LIB

`_send_ping`

```
int _send_ping( longword host, longword countnum, byte ttl,
               byte tos, longword *theid );
```

DESCRIPTION

Generates an ICMP request for host.

PARAMETERS

<code>host</code>	IP address to send ping.
<code>countnum</code>	User-defined count number.
<code>ttl</code>	Time to live for the packets (hop count). 255 is a standard value for this field. See <code>sock_set_ttl()</code> for details.
<code>tos</code>	Type of service on the packets. See <code>sock_set_tos()</code> for details.
<code>theid</code>	The identifier that was sent out.

RETURN VALUE

- 0: Success.
- 1: Failure: unable to resolve hardware address.
- 1: Failure: unable to transmit ICMP request.

LIBRARY

ICMP.LIB

SEE ALSO

`_chk_ping`, `_ping`, `sock_set_ttl`, `sock_set_tos`

setdomainname

```
char *setdomainname( char *name );
```

DESCRIPTION

The domain name returned by `getdomainname()` and used for `resolve()` is set to the value in the string pointed to by `name`. Changing the contents of the string after a `setdomainname()` will change the value of the system domain string. It is not recommended. Instead dedicate a static location for holding the domain name.

`setdomainname(NULL)` is an acceptable way to remove any domain name and subsequent `resolve` calls will not attempt to append a domain name.

PARAMETERS

name Pointer to string.

RETURN VALUE

Pointer to string that was passed in.

LIBRARY

BSDNAME.LIB

SEE ALSO

`getdomainname`, `sethostname`, `gethostname`, `getpeername`,
`getsockname`

sethostid

```
longword sethostid( longword ip );
```

DESCRIPTION

This function changes the system's current IP address. Changing this address will disrupt existing TCP or UDP sessions. You should close all sockets before calling this function.

Normally there is no need to call this function. The macro `MY_IP_ADDRESS` defines an initial IP address for this host, or you can define `USE_DHCP` to obtain a dynamically assigned address. In either case, it is not recommended to use this function to change the address.

PARAMETERS

`ip` New IP address.

RETURN VALUE

New IP address.

LIBRARY

`IP.LIB` (Prior to DC 7.05, this was `DCRTCP.LIB`)

SEE ALSO

[gethostid](#)

sethostname

```
char *sethostname( char *name );
```

DESCRIPTION

Sets the host portion of our name.

PARAMETERS

name Pointer to the new host name.

RETURN VALUE

Pointer to internal hostname buffer on success.

NULL on error (if hostname is too long).

LIBRARY

BSDNAME.LIB

sock_abort

```
void sock_abort( void *s );
```

DESCRIPTION

Close a connection immediately. Under TCP this is done by sending a RST (reset).

Under UDP there is no difference between `sock_close()` and `sock_abort()`.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_close](#), [tcp_open](#)

sock_alive

```
int sock_alive( tcp_socket *s );
```

DESCRIPTION

This function performs the same test as `tcp_tick(s)` i.e., it checks the status of the socket and returns 0 if the socket is fully closed.

The processing overhead of `tcp_tick()` is avoided for cases where several sockets need to be checked in succession.

When this function returns zero for a socket, the socket is then ready for a new call to `tcp_open()` or `tcp_listen()` and friends.

PARAMETER

s TCP socket pointer.

RETURN VALUE

0: Connection reset or fully closed. Socket ready for re-use in another connection.

!0: Connection is opening, established, listening, or in the process of closing.

LIBRARY

NET.LIB

SEE ALSO

`tcp_open`, `tcp_listen`, `sock_close`, `sock_abort`, `tcp_tick`

sock_aread

```
int sock_aread( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

Read exactly `len` bytes from the socket or, if that amount of data is not yet available, do not read anything. Unlike `sock_fastread()`, this function will never return less than the requested amount of data. This can be useful when the application knows that it will be receiving a fixed amount of data, but does not wish to handle the arrival of only part of the data, as it would have to do if `sock_fastread()` was used.

`len` must be less than or equal to the socket receive buffer size, otherwise `sock_fastread()` must be used.

This function is only valid for TCP sockets. It is available starting with DC 7.30.

PARAMETERS

s	Pointer to a TCP socket.
dp	Buffer to place bytes that are read.
len	Number of bytes to copy to the buffer.

RETURN VALUE

- 1: `len` is greater than the total socket receive buffer size, hence this request could never be satisfied in one call.
- 2: The socket is closed or closing, but insufficient data is in the buffer to satisfy the request.
- 3: `len < 0` or the socket parameter was invalid.
- 0: Insufficient data is in the buffer to satisfy the request, or `len` was zero. Try again later since the socket is still able to receive data from the peer.
- `len`: The `len` parameter is returned if there was sufficient data in the socket buffer to satisfy the request.

LIBRARY

TCP.LIB

SEE ALSO

`sock_fastread`, `sock_xfastread`, `sock_fastwrite`,
`sock_xfastwrite`, `sock_axread`, `sock_awrite`, `sock_axwrite`

sock_awrite

```
int sock_awrite( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

Write exactly `len` bytes to the socket or, if that amount of data can not be written, do not write anything. Unlike `sock_fastwrite()`, this function will never return less than the requested amount of data. This can be useful when the application needs to write a fixed amount of data, but does not wish to handle the transmission of only part of the data, as it would have to do if `sock_fastwrite()` was used.

`len` must be less than or equal to the socket transmit buffer size, otherwise `sock_fastwrite()` must be used.

This function is only valid for TCP sockets. It is available starting with DC 7.30.

PARAMETERS

s	Pointer to a TCP socket.
dp	Buffer containing data to write.
len	Number of bytes to write to the socket buffer.

RETURN VALUE

- 1: `len` is greater than the total socket receive buffer size, hence this request could never be satisfied in one call.
- 2: The socket has been closed for further transmissions, e.g., because `sock_close()` has already been called.
- 3: `len < 0` or the socket parameter was invalid.
- 0: Insufficient free space in the transmit buffer to satisfy the request, or `len` was zero. Try again later since the peer will eventually acknowledge the receipt of previous data, freeing up transmit buffer space.
- `len`: The `len` parameter is returned if there was sufficient data in the socket transmit buffer to satisfy the request.

LIBRARY

TCP.LIB

SEE ALSO

`sock_fastread`, `sock_xfastread`, `sock_fastwrite`,
`sock_xfastwrite`, `sock_axread`, `sock_aread`, `sock_axwrite`

sock_axread

```
int sock_axread( tcp_socket *s, long dp, int len );
```

DESCRIPTION

Reads exactly `len` bytes from the socket or, if that amount of data is not yet available, do not read anything.

This function is available starting with DC 7.30. It is identical to `sock_aread()` except that the destination buffer is in `xmem`.

PARAMETERS

s	Pointer to a TCP socket.
dp	Buffer to place bytes that are read.
len	Number of bytes to copy to the buffer.

RETURN VALUE

- 1: `len` is greater than the total socket receive buffer size, hence this request could never be satisfied in one call.
- 2: The socket is closed or closing, but insufficient data is in the buffer to satisfy the request.
- 3: `len < 0` or the socket parameter was invalid.
- 0: Insufficient data is in the buffer to satisfy the request, or `len` was zero. Try again later since the socket is still able to receive data from the peer.
- `len`: The `len` parameter is returned if there was sufficient data in the socket buffer to satisfy the request.

LIBRARY

TCP.LIB

SEE ALSO

`sock_fastread`, `sock_xfastread`, `sock_fastwrite`,
`sock_xfastwrite`, `sock_aread`, `sock_awrite`, `sock_axwrite`

sock_axwrite

```
int sock_axwrite( tcp_socket *s, long dp, int len );
```

DESCRIPTION

Write exactly `len` bytes to the socket or, if that amount of data can not be written, do not write anything. This function is available starting with DC 7.30. It is identical to `sock_awrite()` except that the source buffer is in `xmem`.

Parameters

s	Pointer to a TCP socket.
dp	Buffer containing data to write.
len	Number of bytes to write to the socket buffer.

RETURN VALUE

- 1: `len` is greater than the total socket receive buffer size, hence this request could never be satisfied in one call.
- 2: The socket has been closed for further transmissions, e.g., because `sock_close()` has already been called.
- 3: `len < 0` or the socket parameter was invalid.
- 0: Insufficient free space in the transmit buffer to satisfy the request, or `len` was zero. Try again later since the peer will eventually acknowledge the receipt of previous data, freeing up transmit buffer space.
- `len`: The `len` parameter is returned if there was sufficient data in the socket transmit buffer to satisfy the request.

LIBRARY

TCP.LIB

SEE ALSO

`sock_fastread`, `sock_xfastread`, `sock_fastwrite`,
`sock_xfastwrite`, `sock_axread`, `sock_aread`, `sock_awrite`

sock_bytesready

```
int sock_bytesready( void *s );
```

DESCRIPTION

For TCP sockets:

If the socket is in binary mode, `sock_bytesready()` returns the number of bytes waiting to be read. If there are no bytes waiting, it returns -1.

In ASCII mode, `sock_bytesready()` returns -1 if there are no bytes waiting to be read or the line that is waiting is incomplete (no line terminating character has been read). The number of bytes waiting to be read will be returned given one of the following conditions:

- the buffer is full
- the socket has been closed (no line terminating character can be sent)
- a complete line is waiting

In ASCII mode, a blank line will be read as a complete line with length 0, which will be the value returned. `sock_bytesready()` handles ASCII mode sockets better than `sock_dataready()`, since it can distinguish between an empty line on the socket and an empty buffer.

For UDP sockets:

Returns the number of bytes in the next datagram to be read. If it is a datagram with no data (an empty datagram), then it will return 0. If there are no datagrams waiting, then it returns -1.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

-1: No bytes waiting to be read.

0: If in ASCII mode and a blank line is waiting to be read;
for DC 7.05 and later, a UDP datagram with 0 bytes of data is waiting to be read.

>0: The number of bytes waiting to be read.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_established](#), [sockstate](#)

sock_close

```
void sock_close( void *s );
```

DESCRIPTION

Close an open socket. The socket cannot be reused until it is completely closed.

In the case of UDP, the socket is closed immediately. TCP, being a connection-oriented protocol, must negotiate the close with the remote computer. You can tell a TCP socket is closed by `tcp_tick(s)==NULL` or by running `sock_wait_closed(s)`.

In emergency cases, it is possible to abort the TCP connection rather than close it. Although not recommended for normal transactions, this service is available and is used by all TCP/IP systems.

PARAMETERS

s Pointer to a socket.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_abort](#), [sock_tick](#), [sock_wait_closed](#), [tcp_open](#), [udp_open](#)

sock_dataready

```
int sock_dataready( void *s );
```

DESCRIPTION

Returns the number of bytes waiting to be read. If the socket is in ASCII mode, this function returns zero if a newline character has not been read or the buffer is not full. For UDP sockets, the function returns the number of bytes in the next datagram.

This function cannot tell the difference between no bytes to read and either a blank line or a UDP datagram with no data. For this reason, use `sock_bytesready()` instead.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

0: No bytes to read;
 or newline not yet read if the socket is in ASCII mode;
 or (for DC 7.05 and later) if a UDP datagram has 0 bytes of data waiting to be read.
>0: Number of bytes ready to read.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_bytesready](#)

sockerr

```
char *sockerr( void *s );
```

DESCRIPTION

Gets the last ASCII error message recorded for the specified socket. Use of this function will introduce a lot of string constants in root memory. For production programs, it is better to use error numbers (without translation to strings).

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Pointer to the string that represents the last error message for the socket.

NULL pointer if there have been no errors.

If the symbol `SOCKERR_NO_RETURN_NULL` is defined, then if no error occurred the string "OK" will be returned instead of a NULL pointer.

The error messages are read-only; do not modify them!

LIBRARY

`NETERRNO.LIB`

SEE ALSO

[sock_error](#), [sock_perror](#)

EXAMPLE

```
char *p;
...
if ( p = sockerr(s) )
    printf("Socket closed with error '%s'\n\r", p );
```

`sock_error`

```
int sock_error( void *s, int clear );
```

DESCRIPTION

Return the most recent error number for the specified socket, which may be a TCP or UDP socket. Up to two error codes may be queued to a socket.

PARAMETERS

<code>s</code>	socket
<code>clear</code>	0: do not clear the returned error condition. 1: clear the returned error from the socket. You can call this function again to get the next older error code (if any).

RETURN VALUE

0: No error.

!0: One of the `NETERR_*` constants defined in `NETERRNO.LIB`.

LIBRARY

`NETERRNO.LIB`

SEE ALSO

[sockerr](#), [sock_perror](#)

sock_established

```
int sock_established( void *s );
```

DESCRIPTION

TCP connections require a handshaked open to ensure that both sides recognize a connection. Whether the connection was initiated with `tcp_open()` or `tcp_listen()`, `sock_established()` will continue to return 0 until the connection is established, at which time it will return 1. It is not enough to spin on this after a listen because it is possible for the socket to be opened, written to and closed between two checks. `sock_bytesready()` can be called with `sock_established()` to handle this case.

UDP is a connectionless protocol, hence `sock_established()` always returns 1 for UDP sockets.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

0: Not established.

1: Established.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_bytesready](#), [sockstate](#)

sock_fastread

```
int sock_fastread( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

Reads up to `len` bytes from `dp` on socket `s`. If possible this function fills the buffer, otherwise only the number of bytes immediately available, if any, are returned.

Starting with Dynamic C 7.05, this function is only valid for TCP sockets. For UDP sockets, use `udp_recv()` or `udp_recvfrom()`. Prior to 7.05, this function cannot be used on UDP sockets after `sock_recv_init()` is called.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to put bytes that are read.
len	Maximum number of bytes to write to the buffer.

RETURN VALUE

≥0: Success, number of bytes read.
-1: Error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

`sock_read`, `sock_fastwrite`, `sock_write`, `sockerr`, `udp_recv`,
`udp_recvfrom`, `sock_xfastwrite`, `sock_aread`, `sock_axread`

EXAMPLE

Note that `sock_fastread()` and `sock_read()` do not necessarily return a complete or single line; they return blocks of bytes. In comparison, `sock_getc()` returns a single byte at a time and thus yields poor performance.

```
do {
    /* this function does not block */
    len = sock_fastread(s, buffer, sizeof(buffer)-1);
    if (len>0) {
        buffer[len] = 0;
        printf("%s", buffer);
    }
} while(tcp_tick(s));
```

sock_fastwrite

```
int sock_fastwrite( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

Writes up to `len` bytes from `dp` to socket `s`. This function writes as many bytes as possible to the socket and returns that number of bytes. Starting with Dynamic C 7.05, this function is only valid for TCP sockets. For UDP sockets, use `udp_send()` or `udp_sendto()`.

When using a UDP socket prior to DC 7.05, `sock_fastwrite()` will send one record if

$$\text{len} \leq \text{ETH_MTU} - 20 - 8$$

`ETH_MTU` is the Ethernet Maximum Transmission Unit; 20 is the IP header size and 8 is the UDP header size. By default, this is 572 bytes. If `len` is greater than this number, then the function does not send the data and returns -1. Otherwise, the UDP datagram would need to be fragmented.

For TCP, the new data is queued for sending and `sock_fastwrite()` returns the number of bytes that will be sent. The data may be transmitted immediately if enough data is in the buffer, or sufficient time has expired, or the user has explicitly used `sock_flushnext()` to indicate this data should be flushed immediately. In either case, no guarantee of acceptance at the other end is possible.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to be written.
len	Maximum number of bytes to write to the socket.

RETURN VALUE

≥0: Success, number of bytes written.
-1: Error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

`sock_write`, `sock_fastread`, `sock_read`, `sockerr`, `sock_flush`,
`sock_flushnext`, `udp_send`, `udp_sendto`, `sock_xfastwrite`

sock_flush

```
void sock_flush( tcp_socket *s );
```

DESCRIPTION

`sock_flush()` will flush the unwritten portion of the TCP buffer to the network. No guarantee is given that the data was actually delivered. In the case of a UDP socket, no action is taken.

`sock_flushnext()` is recommended over `sock_flush()`.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

None.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_flushnext](#), [sock_fastwrite](#), [sock_write](#), [sockerr](#)

sock_flushnext

```
void sock_flushnext( tcp_socket *s );
```

DESCRIPTION

Writing to TCP sockets does not guarantee that the data are actually transmitted or that the remote computer will pass that data to the other client in a timely fashion. Using a flush function will guarantee that DCRTCP.LIB places the data onto the network. No guarantee is made that the remote client will receive that data.

`sock_flushnext()` is the most efficient of the flush functions. It causes the next function that sends data to the socket to flush, meaning the data will be transmitted immediately.

Several functions imply a flush and do not require an additional flush: `sock_puts()`, and sometimes `sock_putc()` (when passed a `\n`).

PARAMETERS

s Pointer to a socket.

RETURN VALUE

None.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

`sock_write`, `sock_fastread`, `sock_read`, `sockerr`, `sock_flush`,
`sock_flushnext`

sock_getc

```
int sock_getc( tcp_socket *s );
```

DESCRIPTION

Gets the next character from the socket. NOTE: This function blocks. Starting with Dynamic C 7.05, this function is only valid with TCP sockets. Prior to 7.05, this function could not be used on UDP sockets after `sock_recv_init()` was called.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Character read or -1 if error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_putc](#), [sock_gets](#), [sock_puts](#), [sock_read](#), [sock_write](#)

EXAMPLE

```
do {
    if (sock_bytesready(s) > 0)
        putchar(sock_getc(s));
} while (tcp_tick(s));
```

sock_gets

```
int sock_gets( tcp_socket * s, char * dp, int len );
```

DESCRIPTION

Read a string from a socket and replace the CR or LF with a '\0'. If the string is longer than len, the string is null terminated and any remaining characters will be returned on the next sock_gets() call. This function is only valid for TCP sockets.

To use sock_gets(), you must first set ASCII mode using the function sock_mode() or the macro tcp_set_ascii().

PARAMETERS

s	Pointer to a socket
dp	Buffer to put the string.
len	Max length of buffer.

RETURN VALUE

0: The buffer is empty; or no '\r' or '\n' was read, plus the buffer has room and the connection can get more data.
>0: The length of the string.
-1: Function was called with a UDP socket (valid for Dynamic C 7.05 and later).

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_puts](#), [sock_putc](#), [sock_getc](#), [sock_read](#), [sock_write](#)

EXAMPLE

```
sock_mode( s, TCP_MODE_ASCII );
do {
    if (sock_bytesready( s ) > 0) {
        sock_gets( s, buffer, sizeof(buffer)-1 );
        puts( buffer );
    }
} while (tcp_tick( s );
```

sock_iface

```
byte sock_iface( void *s );
```

DESCRIPTION

Retrieve the interface number of an open socket. May return `IF_ANY` for unbound sockets.

PARAMETER

s Pointer to open TCP or UDP socket.

RETURN VALUE

Interface number (0..`IF_MAX-1`).

`IF_ANY`: If the socket is unbound.

LIBRARY

`NET.LIB`

SEE ALSO

[tcp_extopen](#), [udp_extopen](#), [tcp_extlisten](#)

sock_init

```
int sock_init( void );
```

DESCRIPTION

This function initializes the packet driver and DCRTCP using the compiler defaults for configuration. This function should be called before using other DCRTCP functions.

The return value indicates if `sock_init()` was successful. If it returns 0, then everything was successful. If it returns 1, then the packet driver initialization failed.

Note that the network interface will not necessarily be available immediately after `sock_init()` is called, even if you are simply using an Ethernet interface with a static configuration. This is especially true if you are using DHCP. If you need to make a network connection directly after calling `sock_init()`, then you will probably want to use code like the following:

```
sock_init();
while (ifpending(IF_DEFAULT) == IF_COMING_UP) {
    tcp_tick(NULL);
}
```

The while loop will not finish until the interface has either completely come up or has failed (see the documentation for `ifpending()` for more information).

If you use `ucos2.lib`, be sure to call `OSInit()` before calling `sock_init()`.

RETURN VALUE

0: OK.

1: Ethernet packet driver initialization failed.

Other: reserved.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_init_or_exit

```
void sock_init_or_exit( int verbose );
```

DESCRIPTION

This is a convenience function intended mainly for sample code which starts a network interface. It is equivalent to the following sequence:

```
if (sock_init() != 0)
    exit(-NETERR_IFDOWN);
while (ifpending(IF_DEFAULT) == IF_COMING_UP)
    tcp_tick(NULL);
if (ifpending(IF_DEFAULT) != IF_UP)
    exit(-NETERR_IFDOWN);
```

In other words, it attempts to initialize the network stack via `sock_init()`. It then waits for the default interface to come active (or fail). If the default interface fails to start, then `exit()` is called.

This function is primarily for debugging and sample code, since there is an indeterminate wait for the interface to start. Production applications may not wish to incur this delay at startup, and also they should handle network errors in a manner other than `exit()` since `exit()` is really only useful in a debug environment.

In particular, if the interface is a Wi-Fi interface, then this function will wait until the interface is associated with an access point. If it appears to wait indefinitely, then there is probably an error in the configuration such as a mis-typed SSID, or incorrect security keys.

NOTE: Don't use this function if you intend the network interface to be down after `sock_init()`.

PARAMETER

verbose	Non-zero to print handy message on success. Message shows IP address and netmask of default interface. Also will print messages at intervals while the interface is in a pending state. This can happen if the interface takes a long time to come up. In the case of an Ethernet interface, if you accidentally leave the cable unplugged then this function will loop forever, printing messages at a default 5 second interval.
----------------	--

RETURN VALUE

None. May exit (-NETERR_IFDOWN) if error.

LIBRARY

NET.LIB

SEE ALSO

`sock_init`, `ifpending`, `tcp_tick`

sock_mode

```
word sock_mode( void *s, word mode );
```

DESCRIPTION

Change some of the socket options. Depending on whether *s* is a TCP or UDP socket, you may pass OR'd combinations of the following flags in the mode parameter. For a TCP socket, only the `TCP_MODE_*` flags are relevant. For a UDP socket, only the `UDP_MODE_*` flags are relevant. Do not use the wrong flags for the given socket type.

It is more convenient, faster, and safer to use the macro equivalent if it is only desired to change one mode at a time. If you use this function, then you must specify the setting of all relevant flags (TCP or UDP). The macros do not do socket locking so, strictly speaking, μ C/OS-II users should call this function rather than use the macros.

TCP MODES:

`TCP_MODE_ASCII` | `TCP_MODE_BINARY` (default)

TCP and UDP sockets are usually in binary mode which means an arbitrary stream of bytes is allowed (TCP is treated as a byte stream and UDP is treated as records filled with bytes.) The default is `TCP_MODE_BINARY`. By changing the mode to `TCP_MODE_ASCII`, some of the `DCR TCP.LIB` functions will see a stream of records terminated with a newline character.

In ASCII mode, `sock_bytesready()` will return -1 until a newline-terminated string is in the buffer or the buffer is full. `sock_puts()` will append a newline to any output. `sock_gets()` (which should only be used in ASCII mode) removes the newline and null terminates the string.

Equivalent Macros: `tcp_set_binary(s)` and `tcp_set_ascii(s)`

`TCP_MODE_NAGLE` (default) | `TCP_MODE_NONAGLE`

The Nagle algorithm may substantially reduce network traffic with little negative effect on a user (In some situations, the Nagle algorithm even improves application performance.) The default is `TCP_MODE_NAGLE`. This mode only affects TCP connections.

Equivalent Macros: `tcp_set_nagle(s)` and `tcp_set_nonagle(s)`

sock_mode (cont'd)

UDP MODES:

UDP_MODE_CHK | UDP_MODE_NOCHK

Checksums are required for TCP, but not for UDP. The default is `UDP_MODE_CHK`. If you are providing a checksum at a higher level, the low-level checksum may be redundant. The checksum for UDP can be disabled by selecting the `UDP_MODE_NOCHK` flag. Note that you do not control whether the remote computer will send checksums. If that computer does checksum its outbound data, `DCRTCP.LIB` will check the received packet's checksum.

Equivalent Macros: `udp_set_chk(s)` and `udp_set_nochk(s)`

UDP_MODE_NOICMP (default) | UDP_MODE_ICMP

Marks this socket for receipt of ICMP error messages. The messages are queued like normal received datagrams, and read using `udp_recvfrom()`, which returns -3 when ICMP messages are returned instead of normal datagrams. Only ICMP messages which are relevant to the current binding of the socket are queued.

Equivalent Macros: `udp_set_noicmp(s)` and `udp_set_icmp(s)`

UDP_MODE_NODICMP (default) | UDP_MODE_DICMP

Marks this socket as the default receiver of ICMP messages which cannot be assigned to a particular UDP socket. This would be used for UDP sockets that are used with many different `sendto` addresses, since the ICMP message may refer to a message sent some time ago (with different destination address than the most recent). Only one UDP socket should be set with this mode.

Equivalent Macros: `udp_set_nodicmp(s)` and `udp_set_dicmp(s)`

PARAMETERS

s	Pointer to a socket.
mode	New mode for specified socket.

RETURN VALUE

Resulting mode flags.

SEE ALSO

[inet_addr](#)

LIBRARY

`NET.LIB` (Prior to DC 7.05, this was `DCRTCP.LIB`)

sock_noflush

```
void sock_noflush( tcp_socket *s );
```

DESCRIPTION

This function prevents the next write to the socket from transmitting a data segment. It needs to be issued before each write function in which it is desired not to transmit. It can be used to make more efficient use of network bandwidth when the Nagle algorithm is turned off for the socket. If Nagle is on, then there is not much benefit to using this function.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

None.

LIBRARY

TCP.LIB

SEE ALSO

[sock_flush](#), [sock_flushnext](#), [sock_fastwrite](#), [sock_write](#)

sock_perror

```
void sock_perror( void *s, const char *prefix );
```

DESCRIPTION

Prints out the most recent error messages for a socket, and clear the errors. This calls `sockerr()` and `printf()`, so it should only be called for debugging a new application. The output is in the format:

```
[TCP|UDP] socket (ipaddr:port -> ipaddr:port) msg1; msg2
```

where `msg1` and, possibly, `msg2` are the most recent error messages. The initial string is "TCP" or "UDP" for open sockets, or may be "Closed" if the socket is currently closed (either TCP or UDP). Up to two error codes may be queued to a socket.

If there are no errors, nothing is printed.

PARAMETERS

s	Pointer to TCP or UDP socket.
prefix	Pointer to text to add to generated messages, or NULL.

LIBRARY

NETERRNO.LIB

SEE ALSO

[sock_error](#), [sockerr](#)

sock_preread

```
int sock_preread( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

This function reads up to `len` bytes from the socket into the buffer `dp`. The bytes are not removed from the socket's buffer. This function is only valid with TCP sockets.

PARAMETERS

s	Pointer to a socket structure.
dp	Buffer to preread into.
len	Maximum number of bytes to preread.

RETURN VALUE

0: No data waiting.
-1: Error.
>0: Number of preread bytes.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_fastread](#), [sock_fastwrite](#), [sock_read](#), [sock_write](#)

`sock_putc`

```
byte sock_putc( tcp_socket *s, byte c );
```

DESCRIPTION

A single character is placed on the output buffer. In the case of ‘\n’, the buffer is flushed as described under `sock_flushnext`. No other ASCII character expansion is performed.

Note that `sock_putc` uses `sock_write`, and thus may block if the output buffer is full. See `sock_write` for more details.

Starting with Dynamic C 7.05, this function is only valid with TCP sockets.

PARAMETERS

s	Pointer to a socket.
c	Character to send.

RETURN VALUE

The character `c`.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_read](#), [sock_write](#), [sock_fastread](#), [sock_fastwrite](#), [sock_mode](#)

sock_puts

```
int sock_puts( tcp_socket *s, byte *dp );
```

DESCRIPTION

A string is placed on the output buffer and flushed as described under `sock_flushnext()`. If the socket is in ASCII mode, CR and LF are appended to the string. No other ASCII character expansion is performed. In binary mode, the string is sent as is.

Note that `sock_puts()` uses `sock_write()`, and thus may block if the output buffer is full. See `sock_write()` for more details.

Starting with Dynamic C 7.05, this function is only valid with TCP sockets.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to read the string from.

RETURN VALUE

≥0: Length of string in `dp`.

-1: Function was called with a UDP socket (valid for Dynamic C 7.05 and later).

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

`sock_gets`, `sock_putc`, `sock_getc`, `sock_read`, `sock_write`

`sock_rleft`

```
int sock_rleft( void *s );
```

DESCRIPTION

Determines the number of bytes available in the receive buffer.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Number of bytes available in the receive buffer.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_rbsize](#), [sock_rbused](#), [sock_tbsize](#), [sock_tbused](#), [sock_tbleft](#)

`sock_rbsize`

```
int sock_rbsize( void *s );
```

DESCRIPTION

Determines the size of the receive buffer for the specified socket.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

The size of the receive buffer.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_rbleft](#), [sock_rbused](#), [sock_tbsize](#), [sock_tbused](#), [sock_tbleft](#)

sock_rused

```
int sock_rused( void *s );
```

DESCRIPTION

Returns the number of bytes in use in the receive buffer for the specified socket.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Number of bytes in use.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_rleft](#), [sock_tbsize](#), [sock_tused](#), [sock_tleft](#)

sock_read

```
int sock_read( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

Reads up to `len` bytes from `dp` on socket `s`. This function will busy wait until either `len` bytes are read or there is an error condition. If `sock_yield()` has been called, the user-defined function that is passed to it will be called in a tight loop while `sock_read()` is busy waiting.

Starting with Dynamic C 7.05, this function is only valid for TCP sockets. For UDP sockets, use `udp_recv()` or `udp_recvfrom()`. Prior to 7.05, this function cannot be used on UDP sockets after `sock_recv_init()` is called.

PARAMETERS

s	Pointer to a socket.
dp	Buffer to store bytes that are read.
len	Maximum number of bytes to write to the buffer.

RETURN VALUE

≥0: Success, number of bytes read.
-1: Error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_fastread](#), [sock_fastwrite](#), [sock_write](#), [sockerr](#), [udp_recv](#),
[udp_recvfrom](#)

EXAMPLE

Note that `sock_fastread()` and `sock_read()` do not necessarily return a complete or single line—they return blocks of bytes. In comparison, `sock_getc()` returns a single byte at a time and thus yields poor performance.

```
do {
    len = sock_bytesready(s);
    if (len > 0) {
        if (len > sizeof( buffer) - 1) // If too many bytes, read some
            len = sizeof( buffer) - 1; // now, read the rest next time.

        sock_read( s, buffer, len);
        buffer[len] = 0;
        printf( "%s", buffer);
    }
} while ( tcp_tick(s));
```

sock_readable

```
int sock_readable( void * s );
```

DESCRIPTION

This function determines whether a socket may have data read from it using, for example, `sock_fastread()` or `udp_recvfrom()`.

The parameter may be either a TCP socket or a UDP socket.

The return value is more than a simple boolean: it also indicates the amount of data the socket is guaranteed to deliver with a `sock_fastread()` call that immediately follows (provided that the buffer length is at least that long).

Note: a TCP socket may be readable after it is closed, since there may be pending data in the buffer that has not been read by the application, and it is also possible for the peer to keep sending data.

PARAMETERS

s TCP or UDP socket pointer.

RETURN VALUE

If parameter is a TCP socket (`tcp_Socket *`):

0: socket is not readable. It was aborted by the application or the peer has closed the socket and all pending data has been read by the application. This can be used as a definitive EOF indication for a receive stream.

non-zero: the socket is readable. The amount of data that the socket would deliver is this value minus 1; which may turn out to be zero if the socket's buffer is temporarily empty, or the socket is not yet connected to a peer.

If parameter is a UDP socket (`udp_Socket *`):

0: socket is not open.

non-zero: socket is open. This value minus 1 equals the size of the next datagram in the receive buffer, that would be returned by `udp_recvfrom()` etc. Note that ICMP error messages are also considered if the socket is set up to receive ICMP messages.

LIBRARY

`NET.LIB`

SEE ALSO

`tcp_open`, `tcp_listen`, `sock_close`, `sock_abort`, `tcp_tick`,
`sock_established`, `sock_alive`, `sock_waiting`, `sock_writable`,
`udp_open`, `udp_recvfrom`

sock_recv

```
int sock_recv( sock_type *s, char *buffer, int len );
```

DESCRIPTION

After a UDP socket is initialized with `udp_open()` and `sock_recv_init()`, `sock_recv()` scans the buffers for any datagram received by that socket.

This function is not available starting with Dynamic C 7.05 (see Section 3.5).

PARAMETERS

s	Pointer to a UDP socket.
buffer	Buffer to put datagram.
maxlength	Length of buffer.

RETURN VALUE

>0: Length of datagram.

0: No datagram found.

-1: Receive buffer not initialized with `sock_recv_init()`.

LIBRARY

DCRTCP.LIB

SEE ALSO

`sock_recv_from`, `sock_recv_init`

EXAMPLE USING SOCK_RECV()

```
// Old way of setting network addresses are commented out
//#define MY_IP_ADDRESS "10.10.6.100"
//#define MY_NETMASK "255.255.255.0"

// New way of setting network addresses.
#define TCPCONFIG 1

#mmap xmem

#include "dcrtcp.lib"
#define SAMPLE 401

udp_Socket data;
char bigbuf[8192];

main() {
    word templen;
    char spare[1500];

    sock_init();
    if ( !udp_open( &data, SAMPLE, 0xffffffff, SAMPLE, NULL)
        {
        puts("Could not open broadcast socket");
        exit(3);
        }

    /* set large buffer mode */
    if ( sock_recv_init( &data, bigbuf, sizeof( bigbuf ))) {
        puts("Could not enable large buffers");
        exit(3);
    }

    sock_mode( &data, UDP_MODE_NOCHK );    // turn off checksums

    while (1) {
        tcp_tick( NULL );

        if (templen = sock_recv(&data, spare, sizeof(spare )))
        {
            /* something received */
            printf("Got %u byte packet\n", templen );
        }
    }
}
```

`sock_recv_from`

```
int sock_recv_from( sock_type *s, long *hisip, word *hisport, char
    *buffer, int len );
```

DESCRIPTION

After a UDP socket is initialized with `udp_open()` and `sock_recv_init()`, `sock_recv_from()` scans the buffers for any datagram received by that socket and identifies the remote host's address.

This function is not available starting with Dynamic C 7.05 (see Section 3.5).

PARAMETERS

<code>s</code>	Pointer to UDP socket.
<code>hisip</code>	IP of remote host, according to UDP header.
<code>hisport</code>	Port of remote host.
<code>buffer</code>	Buffer to put datagram in.
<code>len</code>	Length of buffer.

RETURN VALUE

>0: Length of datagram received.
0: No datagram.
-1: Receive buffer was not initialized with `sock_recv_init()`.

LIBRARY

DCRTCP.LIB

SEE ALSO

`sock_recv`, `sock_recv_init`

`sock_recv_init`

```
int sock_recv_init( sock_type *s, void *space, word len );
```

DESCRIPTION

This function is not available starting with Dynamic C 7.05 (see Section 3.5).

The basic socket reading functions (`sock_read()`, `sock_fastread()`, etc.) are not adequate for all your UDP needs. The most basic limitation is their inability to treat UDP as a record service.

A record service must receive distinct datagrams and pass them to the user program as such. You must know the length of the received datagram and the sender (if you opened in broadcast mode). You may also receive the datagrams very quickly, so you must have a mechanism to buffer them.

Once a socket is opened with `udp_open()`, you can use `sock_recv_init()` to initialize that socket for `sock_recv()` and `sock_recv_from()`. Note that `sock_recv()` and related functions are *incompatible* with `sock_read()`, `sock_fastread()`, `sock_gets()` and `sock_getc()`. Once you have used `sock_recv_init()`, you can no longer use the older-style calls.

`sock_recv_init()` installs a large buffer area which gets segmented into smaller buffers. Whenever a UDP datagram arrives, `DCRTCP.LIB` stuffs that datagram into one of these new buffers. The new functions scan those buffers. You must select the size of the buffer you submit to `sock_recv_init()`; make it as large as possible, say 4K, 8K or 16K.

For a sample program, see Example using `sock_recv()` listed under `sock_recv()`.

PARAMETERS

s	Pointer to a UDP socket.
space	Buffer of temporary storage space to store newly received packets.
len	Size of the buffer.

RETURN VALUE

0

LIBRARY

`DCRTCP.LIB`

SEE ALSO

[sock_recv_from](#), [sock_recv](#)

sock_resolved

```
int sock_resolved( void *s );
```

DESCRIPTION

Check whether the socket has a valid destination hardware address. This is typically used for UDP sockets, but may also be used for TCP sockets. If this function returns zero (FALSE), then any datagrams you send using `udp_send()` or `udp_sendto()` may not be transmitted because the destination hardware address is not known.

If the current destination IP address of the socket is zero (i.e., the socket is passively opened), this function returns zero, since datagrams cannot be transmitted from a passively opened socket.

If `udp_bypass_arp()` is in effect, the return value from this function is unaffected, however datagrams will still be sent to the specified hardware address (since the normal resolve process is bypassed).

Note that a hardware address may become invalid after being valid, since the underlying ARP table may need to purge the entry. This would be rare, but if any UDP application needs to ensure that all packets are actually transmitted, which is a questionable goal since UDP is unreliable, then this function should be consulted before each send. If this function returns 0, then the UDP socket should be re-opened.

The hardware address may also be invalidated if `udp_sendto()` is called with a different destination IP address, that has not been determined based on an incoming datagram.

This function is not required for TCP sockets, since the TCP library handles these details internally.

PARAMETER

s Pointer to open TCP or UDP socket

RETURN VALUE:

- 0: Destination hardware address not valid.
- !0: Destination hardware address resolved OK.

LIBRARY

NET.LIB

SEE ALSO

`udp_extopen`, `arpresolve_start`, `arpresolve_check`, `udp_waitopen`,
`udp_sendto`, `udp_bypass_arp`

sock_set_tos

```
void sock_set_tos( void *s, byte tos );
```

DESCRIPTION

Set the IP “Type Of Service” field in outgoing packets for this socket. The given TOS will be in effect until the socket is closed. When a socket is opened (or re-opened), the TOS will be set to the default (TCP_TOS or UDP_TOS as appropriate). If not overridden, the defaults are zero (IPTOS_DEFAULT) in both cases.

PARAMETERS

- | | |
|------------|---|
| s | Pointer to open TCP or UDP socket. |
| tos | Type Of Service. This should be one of the following values: <ul style="list-style-type: none">• IPTOS_DEFAULT - Default service• IPTOS_CHEAP - Minimize monetary cost• IPTOS_RELIABLE - Maximize reliability• IPTOS_CAPACIOUS - Maximize throughput• IPTOS_FAST - Minimize delay• IPTOS_SECURE - Maximize security. |

Other value may be used (since TOS is just a number between 0 and 255), but this should only be done for experimental purposes.

LIBRARY

NET.LIB

SEE ALSO

[sock_set_ttl](#)

sock_set_ttl

```
void sock_set_ttl( void *s, byte ttl );
```

DESCRIPTION

Set the IP “Time To Live” field in outgoing packets for this socket. The given TTL will be in effect until the socket is closed. When a socket is opened (or re-opened), the TTL will be set to the default (TCP_TTL or UDP_TTL as appropriate). If not overridden, the defaults are 64 in both cases.

PARAMETERS

s	Pointer to open TCP or UDP socket.
ttl	Time To Live. This is a value between 1 and 255. A value of zero is also accepted, but will have undesirable consequences.

LIBRARY

NET.LIB

SEE ALSO

[sock_set_tos](#)

sockstate

```
char *sockstate( void *s );
```

DESCRIPTION

Returns a string that gives the current state for a socket.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

An ASCII message which represents the current state of the socket. These strings should not be modified.

"Listen" indicates a passively opened socket that is waiting for a connection.

"SynSent" and "SynRcvd" are connection phase intermediate states.

"Established" states that the connection is complete.

"EstClosing" "FinWait1" "FinWait2" "CloseWait" "Closing" "LastAck" "TimeWait" and "CloseMSL" are connection termination intermediate states.

"Closed" indicates that the connection is completely closed.

"UDP Socket" is always returned for UDP sockets because they are stateless.

"Not an active socket" is a default value used when the socket is not recognized as UDP or TCP.

"BAD" more than one bit set.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_established](#), [sock_dataready](#)

EXAMPLE

```
char *p;
...
#ifdef DEBUG
if ( p = sockstate(s) )
    printf("Socket state is '%s'\n\r", p );
#endif DEBUG
```

sock_tleft

```
int sock_tleft( void *s );
```

DESCRIPTION

Gets the number of bytes left in the transmit buffer. If you do not wish to block, you may first query how much space is available for writing by calling this function before generating data that must be transmitted. This removes the need for your application to also buffer data.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Number of bytes left in the transmit buffer.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_rbsize](#), [sock_rbused](#), [sock_rbleft](#), [sock_tbsize](#), [sock_tbused](#)

EXAMPLE

```
if ( sock_tleft(s) > 10 ) {
    /* we can send up to 10 bytes without blocking or overflowing */
    ...
}
```

`sock_tbsize`

```
int sock_tbsize( void *s );
```

DESCRIPTION

Determines the size of the transmit buffer for the specified socket.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

The size of the transmit buffer.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_rbsize](#), [sock_rbused](#), [sock_rbleft](#), [sock_tbleft](#), [sock_tbused](#)

sock_tbused

```
int sock_tbused( void *s );
```

DESCRIPTION

Gets the number of bytes in use in the transmit buffer for the specified socket.

PARAMETERS

s Pointer to a socket.

RETURN VALUE

Number of bytes in use.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_rbsize](#), [sock_rbused](#), [sock_rbleft](#), [sock_tbsize](#), [sock_tbleft](#)

sock_tick

```
void sock_tick( void *s, int *optional_status_ptr );
```

DESCRIPTION

This macro calls `tcp_tick()` to quickly check incoming and outgoing data and to manage all the open sockets. If our particular socket, `s`, is either closed or made inoperative due to an error condition, `sock_tick()` sets the value of `*optional_status_ptr` (if the pointer is not NULL) to 1, then jumps to a local, user-supplied label, `sock_err`. If the socket connection is fine and the pointer is not NULL `*optional_status_ptr` is set to 0.

PARAMETERS

<code>s</code>	Pointer to a socket.
<code>optional_status_ptr</code>	Pointer to status word.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_wait_closed

```
void sock_wait_closed( void *s, int seconds, int (*fptr)(),
    int *status );
```

DESCRIPTION

This macro waits until a TCP connection is fully closed. Returns immediately for UDP sockets. On an error, the macro jumps to a local, user-supplied `sock_err` label. If `fptr` returns non-zero the macro returns with the status word set to the value of `fptr`'s return value.

This macro has been deprecated in Dynamic C version 7.20.

PARAMETERS

s	Pointer to a socket.
seconds	Number of seconds to wait before timing out. A value of zero indicates the macro should never time-out. A good value to use is <code>sock_delay</code> , a system variable set on configuration. Typically <code>sock_delay</code> is about 20 seconds, but can be set to something else in <code>main()</code> .
fptr	Function to call repeatedly while waiting. This is a user-supplied function.
status	Pointer to a status word.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_wait_established

```
void sock_wait_established( void *s, int seconds, int (*fptr)(),
    int *status );
```

DESCRIPTION

This macro waits until a connection is established for the specified TCP socket, or aborts if a time-out occurs. It returns immediately for UDP sockets. On an error, the macro jumps to the local, user-supplied `sock_err` label. If `fptr` returns non-zero, the macro returns.

This macro has been deprecated in Dynamic C version 7.20.

PARAMETERS

s	Pointer to a socket.
seconds	Number of seconds to wait before timing out. A value of zero indicates the macro should never time-out. A good value to use is <code>sock_delay</code> , a system variable set on configuration. Typically <code>sock_delay</code> is about 20 seconds, but can be set to something else in <code>main()</code> .
fptr	Function to call repeatedly while waiting. This is a user-supplied function.
status	Pointer to a status word.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_waiting

```
int sock_waiting( tcp_socket * s);
```

DESCRIPTION

This function determines whether a TCP socket is waiting for a connection establishment. It returns TRUE (non-zero) if and only if the socket is open, but not YET established.

The purpose of this function is to simplify the application logic in programs which interleave TCP/IP functions with other processing i.e., “non-blocking” style.

NOTE: it is an error to pass a UDP socket to this function. UDP sockets are connectionless, so there is no concept of “waiting for a connection.”

PARAMETER

s TCP socket pointer. This should be a TCP socket which was opened using `tcp_listen()`, `tcp_extlisten()`, `tcp_open()` or `tcp_extopen()`.

RETURN VALUE

0: socket is not waiting. In this case, then next tests that the application should perform are:

- a. `sock_established()`: if this returns TRUE, a connection is currently established. The application can now communicate using `sock_read()`, `sock_write()` etc., then finally call `sock_close()`.
- b. `sock_alive()`: if this returns FALSE, then the socket was aborted by the peer. The application may re-open or re-listen the socket.
- c. Otherwise, the socket was established, but is now closing because the peer closed its side of the connection. The application MAY be able to read and/or write to the socket (depending on protocol) however the amount of readable data will be limited. The application should call `sock_close()` or `sock_abort()`.

In cases (a) and (c), a socket should not be re-opened until `tcp_tick()` on that socket returns 0.

Note that '0' is returned for invalid sockets (e.g., UDP sockets or sockets that are closed).

non-zero: the socket is waiting for a connection. The application should keep calling `tcp_tick()` until this function returns 0.

LIBRARY

NET.LIB

SEE ALSO

`tcp_open`, `tcp_listen`, `sock_close`, `sock_abort`, `tcp_tick`,
`sock_established`, `sock_alive`

sock_wait_input

```
void sock_wait_input( void *s, int seconds, int (*fptr)(),
    int *status );
```

DESCRIPTION

Waits until input exists for functions such as `sock_read()` and `sock_gets()`. As described under `sock_mode()`, if in ASCII mode, `sock_wait_input` only returns when a complete string exists or the buffer is full. It returns immediately for UDP sockets.

On an error, the macro jumps to a local, user-supplied `sock_err` label. If `fptr` returns non-zero, the macro returns.

This macro has been deprecated in Dynamic C version 7.20.

PARAMETERS

s	Pointer to a socket.
seconds	Number of seconds to wait before timing out. A value of zero indicates the macro should never time-out. A good value to use is <code>sock_delay</code> , a system variable set on configuration. Typically <code>sock_delay</code> is about 20 seconds, but can be set to something else in <code>main()</code> .
fptr	Function to call repeatedly while waiting.
status	A pointer to the status word. If this parameter is <code>NULL</code> , no status number will be available, but the macro will otherwise function normally. Typically the pointer will point to a local signed integer that is used only for status. <code>status</code> may be tested to determine how the socket was ended. A value of 1 means a proper close while a -1 value indicates a reset or abort.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

sock_writable

```
int sock_writable( void * s );
```

DESCRIPTION

This function determines whether a socket may have data written to it using (e.g.) `sock_fastwrite()` or `udp_sendto()`.

The parameter may be either a TCP socket or a UDP socket.

The return value is more than a simple boolean: it also indicates the amount of data the socket is guaranteed to accept with a `sock_fastwrite()` call that immediately follows.

NOTE: a TCP socket may be writable before it is established. In this case, any written data is transferred as soon as the connection is established.

PARAMETER

s TCP or UDP socket pointer.

RETURN VALUE

If parameter is a TCP socket (`tcp_Socket *`):

0: socket is not writable. It was closed by the application or it may have been aborted by the peer.

non-zero: the socket is writable. The amount of data that the socket would accept is this value minus 1; which may turn out to be zero if the socket's buffer is temporarily full. On a freshly-established socket, and at any other time when all data has been acknowledged by the peer, the return value (minus one) indicates the maximum socket transmit buffer size.

If parameter is a UDP socket (`udp_Socket *`):

0: socket is not open.

non-zero: socket is open. This value minus 1 equals the maximum size datagram payload that would be sent without fragmentation at the IP level.

Note: the maximum payload depends on the interface that is selected. Since this is not known a priori, the interface with the largest MTU is arbitrarily selected.

LIBRARY

NET.LIB

SEE ALSO

`tcp_open`, `tcp_listen`, `sock_close`, `sock_abort`, `tcp_tick`,
`sock_established`, `sock_alive`, `sock_waiting`, `sock_readable`,
`udp_open`, `udp_sendto`

sock_write

```
int sock_write( tcp_socket *s, byte *dp, int len );
```

DESCRIPTION

Writes up to `len` bytes from `dp` to socket `s`. This function busy waits until either the buffer is completely written or a socket error occurs. If `sock_yield()` has been called, the user-defined function that is passed to it will be called in a tight loop while `sock_write()` is busy-waiting.

For UDP, `sock_write()` will send one (or more) records. For TCP, the new data may be transmitted if enough data is in the buffer or sufficient time has expired or the user has explicitly used `sock_flushnext()` to indicate this data should be flushed immediately. In either case, there is no guarantee of acceptance at the other end.

Starting with Dynamic C 7.05, this function is only valid for TCP sockets. For UDP sockets, use `udp_send()` or `udp_sendto()`.

PARAMETERS

s	Pointer to a socket.
dp	Pointer to a buffer.
len	Maximum number of bytes to write to the buffer.

RETURN VALUE

Number of bytes written or `-1` on an error.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[sock_read](#), [sock_fastwrite](#), [sock_fastread](#), [sockerr](#), [sock_flush](#), [sock_flushnext](#), [udp_send](#), [udp_sendto](#)

sock_xfastread

```
int sock_xfastread( tcp_socket *s, long dp, long len );
```

DESCRIPTION

Reads up to `len` bytes from `dp` on socket `s`. If possible this function fills the buffer, otherwise only the number of bytes immediately available if any are returned. This function is only valid for TCP sockets. For UDP sockets, use `udp_recv()` or `udp_recvfrom()`.

This function is identical to `sock_fastread()`, except that it reads into an extended memory buffer.

PARAMETERS

s	Pointer to socket.
dp	Buffer to place bytes that are read, as an <code>xmem</code> address obtained from, for example, <code>xalloc()</code> .
len	Maximum number of bytes to write to the buffer.

RETURN VALUE

Number of bytes read or -1 if there was an error.

LIBRARY

TCP.LIB

SEE ALSO

[sock_read](#), [sock_fastwrite](#), [sock_write](#), [sockerr](#), [udp_recv](#), [udp_recvfrom](#), [sock_fastread](#)

sock_xfastwrite

```
int sock_xfastwrite( tcp_socket *s, long dp, long len );
```

DESCRIPTION

Writes up to `len` bytes from `dp` to socket `s`. This function writes as many bytes possible to the socket and returns that number of bytes. This function is only valid for TCP sockets. For UDP sockets, use `udp_send()` or `udp_sendto()`.

This function is identical to `sock_fastwrite()`, except that an extended memory data source is used.

PARAMETERS

s	Pointer to socket.
dp	Buffer containing data to be written, as an xmem address obtained from, for example, <code>xalloc()</code> .
len	Maximum number of bytes to write to the socket.

RETURN VALUE

Number of bytes written or -1 if there was an error.

LIBRARY

TCP.LIB

SEE ALSO

`sock_write`, `sock_fastread`, `sock_read`, `sockerr`, `sock_flush`,
`sock_flushnext`, `udp_send`, `udp_sendto`, `sock_fastwrite`

sock_yield

```
int sock_yield( tcp_socket *s, void (*fn)() );
```

DESCRIPTION

This function, if called prior to one of the blocking functions, will cause `fn`, the user-defined function that is passed in as the second parameter, to be called repeatedly while the blocking function is in a busywait state.

PARAMETERS

s	Pointer to a TCP socket.
fn	User-defined function.

RETURN VALUE

0

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

tcp_clearreserve

```
void tcp_clearreserve( word port );
```

DESCRIPTION

This function causes DCRTCP to handle a socket connection to the specified port normally. This undoes the action taken by `tcp_reserveport()`.

PARAMETERS

`port` Port to use.

RETURN VALUE

None.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

`tcp_open`, `tcp_listen`, `tcp_reserveport`

tcp_config

```
void tcp_config( char *name, char *value );
```

DESCRIPTION

Sets TCP/IP stack parameters at runtime. It should not be called with open sockets.

Note that there are specific (and safer) functions for modifying some of the common parameters.

This function is deprecated and will be removed in a future release. The recommended replacement function is `ifconfig`.

PARAMETERS

name	Setting to be changed. The possible parameters are: MY_IP_ADDRESS: host IP address (use <code>sethostid()</code> instead) MY_NETMASK MY_GATEWAY: host's default gateway MY_NAMESERVER: host's default nameserver MY_HOSTNAME MY_DOMAINNAME: host's domain name (use <code>setdomainname()</code> instead)
value	The value to assign to name.

RETURN VALUE

None.

LIBRARY

NET.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

`tcp_open`, `sock_close`, `sock_abort`, `sethostid`, `setdomainname`,
`sethostname`

tcp_extlisten

```
int tcp_extlisten( tcp_socket *s, int iface, word lport, longword
    remip, word port, dataHandler_t datahandler, word reserved, long
    buffer, int buflen );
```

DESCRIPTION

This function tells DCRTCP that an incoming session for a particular port will be accepted. The `buffer` and `buflen` parameters allow a user to supply a socket buffer, instead of using a socket buffer from the pool. `tcp_extlisten()` is an extended version of `tcp_listen()`.

PARAMETERS

s	Pointer to the socket's data structure.
iface	Local interface on which to open the socket. Use <code>IF_ANY</code> if the socket is to accept connections from any interface. Otherwise, connections will be accepted only from the specified interface. Prior to Dynamic C 7.30 this parameter was not implemented and should be <code>IF_DEFAULT</code> .
lport	Port to listen on.
remip	IP address to accept connections from or 0 for all.
port	Port to accept connections from or 0 for all.
datahandler	Function to call when data is received, <code>NULL</code> for placing data in the socket's receive buffer. Prior to Dynamic C 7.30, some details for implementation of this service had not been finalized. Insert a value of <code>NULL</code> if you are using a version of Dynamic C prior to 7.30.
reserved	Set to 0 for now. This parameter is for compatibility and possible future use.
buffer	Address of user-supplied socket buffer in <code>xmem</code> . This is the return value of <code>xalloc()</code> . If <code>buffer</code> is 0, the socket buffer for this socket is pulled from the buffer pool defined by the macro <code>MAX_TCP_SOCKET_BUFFERS</code> .
buflen	Length of user-supplied socket buffer.

RETURN VALUE

- 0: Failure.
- 1: Success.

LIBRARY

`TCP.LIB`

tcp_extopen

```
int tcp_extopen( tcp_socket *s, int iface, word lport, longword
    remip, word port, dataHandler_t datahandler, long buffer, int
    buflen );
```

DESCRIPTION

Actively creates a session with another machine. The `buffer` and `buflen` parameters allow a user to supply a socket buffer, instead of using a socket buffer from the pool.

`tcp_extopen()` is an extended version of `tcp_open()`.

s	Pointer to socket's data structure.
iface	Local interface on which to open the socket. Use <code>IF_ANY</code> if interface is to be selected automatically based on the destination IP address.
lport	Our port, zero for the next one available in the range 1025-65536.
remip	IP address to connect to.
port	Port to connect to.
datahandler	Function to call when data is received, <code>NULL</code> for placing data in the socket's receive buffer. Prior to Dynamic C 7.30, some details for implementation of this service had not been finalized. Insert a value of <code>NULL</code> if you are using a version of Dynamic C prior to 7.30.
buffer	Address of user-supplied socket buffer in <code>xmem</code> . This is the return value of <code>xalloc()</code> . If <code>buffer</code> is 0, the socket buffer for this socket is pulled from the buffer pool defined by the macro <code>MAX_TCP_SOCKET_BUFFERS</code> .
buflen	Length of user-supplied socket buffer.

RETURN VALUE

0: Error, unable to resolve the remote computer's hardware address.

!0: Success.

LIBRARY

`TCP.LIB`

SEE ALSO

[tcp_open](#)

tcp_keepalive

```
int tcp_keepalive( tcp_socket *s, long timeout );
```

DESCRIPTION

Enable or disable TCP keepalives on a specified socket. The socket must already be open. Keepalives will then be sent after `timeout` seconds of inactivity. It is highly recommended to keep `timeout` as long as possible, to reduce the load on the network. Ideally, it should be no shorter than 2 hours. After the timeout is sent, and `KEEPALIVE_WAITTIME` seconds pass, another keepalive will be sent, in case the first was lost. This will be retried `KEEPALIVE_NUMRETRY`s times. Both of these macros can be defined at the top of your program, overriding the defaults of 60 seconds, and 4 retries.

Using keepalives is not a recommended procedure. Ideally, the application using the socket should send its own keepalives. `tcp_keepalive()` is provided because telnet and a few other network protocols do not have a method of sending keepalives at the application level.

PARAMETERS

s	Pointer to a socket.
timeout	Period of inactivity, in seconds, before sending a keepalive or 0 to turn off keepalives.

RETURN VALUE

0: Success.
1: Failure.

LIBRARY

TCP.LIB

SEE ALSO

[sock_fastread](#), [sock_fastwrite](#), [sock_write](#), [sockerr](#)

tcp_listen

```
int tcp_listen( tcp_socket *s, word lport, longword remip,
               word port, dataHandler_t datahandler, word reserved );
```

DESCRIPTION

This function tells DCRTCP.LIB that an incoming session for a particular port will be accepted. After a call to `tcp_listen()`, the function `sock_established()` (or the macro `sock_wait_established`) must be called to poll the connection until a session is fully established.

It is possible for a connection to be opened, written to and closed between two calls to the function `sock_established()`. To handle this case, call `sock_bytesready()` to determine if there is data to be read from the buffer.

Multiple calls to `tcp_listen()` to the same local port (`lport`) are acceptable and constitute the mechanism for supporting multiple incoming connections to the same local port. Each time another host attempts to open a session on that particular port, another one of the listens will be consumed until such time as all listens have become established sessions and subsequent remote host attempts will receive a reset.

PARAMETERS

s	Pointer to a socket.
lport	Port to listen on (the local port number).
remip	IP address of the remote host to accept connections from or 0 for all.
port	Port to accept connections from or 0 for all.
datahandler	Function to call when data is received; NULL for placing data in the socket's receive buffer. Prior to Dynamic C 7.30, some details for implementation of this service had not been finalized. Insert a value of NULL if you are using a version of Dynamic C prior to 7.30.
reserved	Set to 0 for now. This parameter is for compatibility and possible future use.

RETURN VALUE

- 0: Failure.
- 1: Success.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[tcp_extlisten](#)

EXAMPLE USING TCP_LISTEN()

```
// Old way of setting network addresses is commented out.
//#define MY_IP_ADDRESS "10.10.6.100"
//#define MY_NETMASK "255.255.255.0"

// New method of setting network addresses
#define TCPCONFIG 1

#memmap xmem
#use "dcrtcp.lib"

#define TELNET_PORT 23

static tcp_Socket *s;
char *userid;

telnets(int port) {
    tcp_Socket telnetsock;
    char buffer[512];
    int status;
    s = &telnetsock;
    tcp_listen( s, port, 0L, 0, NULL, 0);
    while (!sock_established(s) && sock_bytesready(s)==-1){
        tcp_tick(NULL);
    }
    puts("Receiving incoming connection");
    sock_mode( s, TCP_MODE_ASCII );
    sock_puts( s, "Welcome to a sample telnet server.");
    sock_puts( s, "Each line you type will be printed on\"
    \" this screen once you hit return.");
    /* other guy closes connection except if we timeout ... */
    do {
        if (sock_bytesready(s) >= 0) {
            sock_gets(s, buffer, sizeof(buffer)-1);
            puts ( buffer);
        }
    } while (tcp_tick(s));
}

main() {
    sock_init();
    telnets( TELNET_PORT);
    exit(0);
}
```

tcp_open

```
int tcp_open( tcp_Socket *s, word lport, longword remip,
              word port, dataHandler_t datahandler );
```

DESCRIPTION

This function actively creates a session with another machine. After a call to `tcp_open()`, the function `sock_established()` (or the macro `sock_wait_established`) must be called to poll the connection until a session is fully established.

It is possible for a connection to be opened, written to and closed between two calls to the function `sock_established()`. To handle this case, call `sock_bytesready()` to determine if there is data to be read from the buffer.

PARAMETERS

s	Pointer to a socket structure.
lport	Our local port. Use zero for the next available port in the range 1025-65536. A few applications will require you to use a particular local port number, but most network applications let you use almost any port with a certain set of restrictions. For example, FINGER or TELNET clients can use any local port value, so pass the value of zero for <code>lport</code> and let <code>DCRTCP.LIB</code> pick one for you.
remip	IP address to connect to.
port	Port to connect to.
datahandler	Function to call when data is received; <code>NULL</code> for placing data in the socket's receive buffer. Prior to Dynamic C 7.30, some details for implementation of this service had not been finalized. Insert a value of <code>NULL</code> if you are using a version of Dynamic C prior to 7.30.

RETURN VALUE

0: Unable to resolve the remote computer's hardware address.
!0 otherwise.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[tcp_listen](#)

EXAMPLE USING TCP_OPEN()

```
// Old way of setting network addresses is commented out.
//#define MY_IP_ADDRESS "10.10.6.100"
//#define MY_NETMASK "255.255.255.0"

// New of setting network addresses
#define TCPCONFIG 1

#memmap xmem
#use "dcrtcp.lib"

#define ADDRESS "10.10.6.19"
#define PORT "200"

main() {
    word status;
    word port;
    longword host;
    tcp_Socket tsock;

    sock_init();

    if (!(host = resolve(ADDRESS))) {
        puts("Could not resolve host");
        exit(3);
    }
    port = atoi( PORT );

    printf("Attempting to open '%s' on port %u\n\r", ADDRESS,
        port );

    if ( !tcp_open( &tsock, 0, host, port , NULL ) ) {
        puts("Unable to open TCP session");
        exit(3);
    }

    printf("Waiting a maximum of %u seconds for connection"\
        " to be established\n\r", sock_delay );

    while (!sock_established(&tsock) &&
        sock_bytesready(&tsock)== -1){
        tcp_tick(NULL);
    }
    puts("Socket is established");
    sock_close( &tsock );
    exit(0);
}
```

tcp_reserveport

```
void tcp_reserveport( word port );
```

DESCRIPTION

This function allows a connection to be established even if there is not yet a socket available. This is done by setting a parameter in the TCP header during the connection setup phase that indicates 0 bytes of data can be received at the present time. The requesting end of the connection will wait until the TCP header parameter indicates that data will be accepted.

The 2MSL waiting period for closing a socket is avoided by using this function.

The penalty of slower connection times on a controller that is processing a large number of connections is offset by allowing the program to have less sockets and consequently less RAM usage.

PARAMETERS

port Port to use.

RETURN VALUE

None.

LIBRARY

TCP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[tcp_open](#), [tcp_listen](#), [tcp_clearreserve](#)

tcp_tick

```
int tcp_tick( void *s );
```

DESCRIPTION

This function is a single kernel routine designed to quickly process packets and return as soon as possible. `tcp_tick()` performs processing on all sockets upon each invocation: checking for new packets, processing those packets, and performing retransmissions on lost data. On most other computer systems and other kernels, performing these required operations in the background is often done by a task switch. `DCRTCP.LIB` does not use a tasker for its basic operation, although it can adopt one for the user-level services.

Although you may ignore the returned value of `tcp_tick()`, it is the easiest method to determine the status of the given socket.

PARAMETERS

s Pointer to a socket. If a `NULL` pointer is passed in the returned value should be ignored.

RETURN VALUE

0: Connection reset or closed by other host or `NULL` was passed in.
!0: Connection is fine.

LIBRARY

`TCP.LIB` (Prior to DC 7.05, this was `DCRTCP.LIB`)

SEE ALSO

[tcp_open](#), [sock_close](#), [sock_abort](#)

udp_bypass_arp

```
void udp_bypass_arp( udp_socket *s, eth_address *eth );
```

DESCRIPTION

Override the normal Address Resolution Protocol for this UDP socket. This is sometimes necessary for special purposes such as if the Ethernet address is to remain fixed, or if the Ethernet address is not obtainable using ARP. The great majority of applications should not use this function.

If ARP bypass is in effect for a UDP socket, then `udp_sendto()` will never return the -2 return code.

The destination interface is also forced to be `IF_DEFAULT`. If the supplied hardware address is accessible from a non-default interface only, then you will need to manually set the `s->iface` field.

PARAMETERS

s	UDP socket
eth	Pointer to override address. If <code>NULL</code> , then resume normal operation i.e., use ARP to resolve Ethernet addresses. Note that the specified Ethernet address must be in static storage, since only the pointer is stored.

LIBRARY

`UDP.LIB`

SEE ALSO

[udp_sendto](#), [udp_waitsend](#), [sock_resolved](#)

udp_close

```
void udp_close( udp_socket *ds );
```

DESCRIPTION

This function closes a UDP connection. Starting with Dynamic C 7.30, this function performs the actions necessary to leave a host group when closing a multicast socket. It is IGMPv2 compliant.

PARAMETERS

ds Pointer to socket's data structure.

LIBRARY

UDP.LIB

udp_extopen

```
int udp_extopen( udp_socket *s, int iface, word lport, longword
    remip, word port, dataHandler_t datahandler, long buffer, int
    buflen );
```

DESCRIPTION

This function is an extended version of `udp_open()`. It opens a socket on a given network interface (`iface`) on a given local port (`lport`). If the remote IP address is specified (`remip`), then only UDP datagrams from that host will be accepted.

The remote end of the connection is specified by `remip` and `port`. The following table explains the possible combinations and what they mean.

REMIP	Effect of REMIP value
0	The connection completes when the first datagram is received, supplying both the remote IP address and the remote port number. Only datagrams received from that IP/port address will be accepted.
-1	All remote hosts can send datagrams to the socket. All outgoing datagrams will be sent to the broadcast address unless <code>udp_sendto()</code> specifies otherwise.
>0	If the remote IP address is a valid IP address and the remote port is 0, the connection will complete when the first datagram is received, supplying the remote port number. If the remote IP address and the remote port are both specified when the function is called, the connection is complete at that point.

The `buffer` and `buflen` parameters allow a user to supply a socket buffer, instead of using a socket buffer from the pool.

If `remip` is non-zero, then the process of resolving the correct destination hardware address is started. Datagrams cannot be sent until `sock_resolved()` returns TRUE. If you attempt to send datagrams before this, then the datagrams may not get sent. The exception to this is if `remip` is -1 (broadcast) in which case datagrams may be sent immediately after calling this function.

This function also works with multicast addresses. If `remip` is a multicast address, then packets sent with this function will go to the multicast address, and packets received will also be from that multicast address. Also, if enabled, IGMP will be used to join the multicast groups. The group will be left when the socket is closed. Note that if `port` is 0 and `remip` is a multicast address, the port will not be filled in on the first received datagram (that is, the socket is non-binding to the port).

udp_extopen (cont.)

PARAMETERS

s	Pointer to socket.
iface	Local interface on which to open the socket. Use <code>IF_ANY</code> if the socket is to accept datagrams from any interface. Otherwise, datagrams will be accepted only from the specified interface. This parameter is supported as of Dynamic C 7.30. With earlier version of DC, this parameter should be <code>IF_DEFAULT</code> .
lport	Local port.
remip	Acceptable remote IP, or 0 for all.
port	Acceptable remote port, or 0 for all.
datahandler	Function to call when data is received, <code>NULL</code> for placing data in the socket's receive buffer.
buffer	Address of user-supplied socket buffer in <code>xmem</code> . If <code>buffer</code> is 0, the socket buffer for this socket is pulled from the buffer pool defined by the macro <code>MAX_UDP_SOCKET_BUFFERS</code> .
buflen	Length of user-supplied socket buffer.

RETURN VALUE:

!0: Success.

0: Failure; error opening socket, e.g., a buffer could not be allocated.

LIBRARY

`UDP.LIB`

SEE ALSO

[udp_open](#), [sock_resolved](#)

udp_open

```
int udp_open( udp_Socket *s, word lport, longword remip,  
             word port, dataHandler_t datahandler );
```

DESCRIPTION

This function opens a UDP socket on the given local port (`lport`). If the remote IP address is specified (`remip`), then only UDP datagrams from that host will be accepted. The remote end of the connection is specified by `remip` and `port`. The following table explains the possible combinations and what they mean.

REMIP	Effect of REMIP value
0	The connection completes when the first datagram is received, supplying both the remote IP address and the remote port number. Only datagrams received from that IP/port address will be accepted.
-1	All remote hosts can send datagrams to the socket. All outgoing datagrams will be sent to the broadcast address on the specified port. The <code>port</code> parameter is ignored.
>0	If the remote IP address is a valid IP address and the remote port is 0, the connection will complete when the first datagram is received, supplying the remote port number. If the remote IP address and the remote port are both specified when the function is called, the connection is complete at that point.

If the remote host is set to a particular address, either host may initiate traffic. Multiple calls to `udp_open()` with `remip` set to zero is a useful way of accepting multiple incoming sessions.

Although multiple calls to `udp_open()` may normally be made with the same `lport` number, only one `udp_open()` should be made on a particular `lport` if the `remip` is set to -1. Essentially, the broadcast and nonbroadcast protocols cannot co-exist.

Be sure that you have allocated enough UDP socket buffers with `MAX_UDP_SOCKET_BUFFERS`. Note that this macro defaults to 0, so any usage of `udp_open()` requires a definition of `MAX_UDP_SOCKET_BUFFERS` in your program.

udp_open (cont.)

This function also works with multicast addresses. If `remip` is a multicast address, then packets sent with this function will go to the multicast address, and packets received will also be from that multicast address. Also, if enabled, IGMP will be used to join the multicast groups. The group will be left when the socket is closed. Note that if `port` is 0 and `remip` is a multicast address, the port will not be filled in on the first received datagram (that is, the socket is non-binding to the port).

PARAMETERS

s	Pointer to a UDP socket.
lport	Local port
remip	Acceptable remote IP, 0 to connect on first datagram, or -1 for broadcast.
port	Acceptable remote port, or 0 to connect on first datagram.
datahandler	Function to call when data is received. NULL for placing data in the socket's receive buffer.

RETURN VALUE

0: Failure (e.g., a buffer could not be allocated).
!0: Success.

LIBRARY

UDP.LIB (Prior to DC 7.05, this was DCRTCP.LIB)

SEE ALSO

[udp_extopen](#)

udp_peek

```
int udp_peek( udp_socket *s, _udp_datagram_info *udi );
```

DESCRIPTION

Look into the UDP socket receive buffer to see if there is a datagram ready to be read using `udp_recvfrom()`. This function does not remove the datagram from the buffer, but it allows the application to determine the full details about the next datagram, including whether the datagram was broadcast.

The returned data is put in `*udi`. `udi` must point to a valid data structure, or be `NULL`. The data structure is:

```
typedef struct {
    longword remip;    // Remote host IP address
    word      remport; // Remote host port number
    int       len;     // Length of datagram
    byte      flags;   // Bit mask (defined below)
    byte      iface;   // Interface number
} _udp_datagram_info;
```

The `flags` field may have one of the following values:

```
UDI_ICMP_ERROR    - This is an ICMP error entry.
UDI_TOS_MASK      - Type-of-service bit mask.
UDI_BROADCAST_LL  - Received on broadcast link layer address.
UDI_BROADCAST_IP  - Received on broadcast network (IP) address.
```

PARAMETERS

s	UDP socket to check
udi	Where to store the returned information.

RETURN VALUE

1: A normal datagram is in the receive buffer.
0: No datagram waiting.
-3: ICMP error message in receive buffer - will only be returned if `udi` is not `NULL`.

LIBRARY

`UDP.LIB`

SEE ALSO

[udp_recvfrom](#)

udp_recv

```
int udp_recv( udp_socket *s, char *buffer, int len );
```

DESCRIPTION

Receives a single UDP datagram on a UDP socket. If the buffer is not large enough for the datagram, the datagram is truncated, and the remainder discarded.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer where the UDP datagram will be stored.
len	Maximum length of the buffer.

RETURN VALUE

≥0: Number of bytes received.
-1: No datagram waiting.
<-1: Error.

LIBRARY

UDP.LIB

SEE ALSO

[udp_recvfrom](#), [udp_send](#), [udp_sendto](#), [udp_open](#)

udp_recvfrom

```
int udp_recvfrom( udp_socket *s, char *buffer, int len,
                 longword *remip, word *remport );
```

DESCRIPTION

Receive a single UDP datagram on a UDP socket. `remip` and `remport` should be pointers to the locations where the remote IP address and remote port from which the datagram originated are placed. If the buffer is not large enough for the datagram, then the datagram will be truncated, with the remainder being discarded.

If and only if the `UDP_MODE_ICMP` or `UDP_MODE_DICMP` modes are set for this socket, then a return code of -3 indicates that an ICMP error message is being returned in the buffer instead of a normal datagram. In this case, `buffer` will contain fixed data in the form of a structure of type `_udp_icmp_message`. The definition of this structure is:

```
typedef struct {
    word myport;           // Originating port on this host
    byte icmp_type;       // One of the ICMP_TYPE_* values
    byte icmp_code;       // The corresponding ICMP code
} _udp_icmp_message;
```

Please see `sock_mode` for more information about the modes `UDP_MODE_ICMP` and `UDP_MODE_DICMP`.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer where the UDP datagram will be stored.
len	Maximum length of the buffer.
remip	IP address of the remote host of the received datagram.
remport	Port number of the remote host of the received datagram.

RETURN VALUE

≥0: Number of bytes received.
-1: No datagram waiting.
-2: Error - not a UDP socket.
-3: The returned buffer contains an ICMP error which was queued previously.

LIBRARY

UDP.LIB

SEE ALSO

[udp_recv](#), [udp_send](#), [udp_sendto](#), [udp_open](#), [udp_peek](#)

udp_send

```
int udp_send( udp_socket *s, char *buffer, int len );
```

DESCRIPTION

Sends a single UDP datagram on a UDP socket. It will not work for a socket for which the `remip` parameter to `udp_open()` was 0, unless a datagram has first been received on the socket. If the `remip` parameter to `udp_open()` was -1, the datagram will be sent to the broadcast address.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer that contains the UDP datagram
len	Length of the UDP datagram.

RETURN VALUE

≥0: Number of bytes sent.
-1: Failure.
-2: Failed because hardware address not resolved.

LIBRARY

UDP.LIB

SEE ALSO

[udp_sendto](#), [udp_recv](#), [udp_recvfrom](#), [udp_open](#)

udp_sendto

```
int udp_sendto( udp_socket *s, char *buffer, int len,
               longword remip, word remport );
```

DESCRIPTION

Sends a single UDP datagram on a UDP socket. It will send the datagram to the IP address and port specified by `remip` and `remport`. Note that this function can be used on a socket that has been "connected" to a different remote host and port.

PARAMETERS

s	Pointer to socket's data structure.
buffer	Buffer that contains the UDP datagram.
len	Length of the UDP datagram.
remip	IP address of the remote host.
remport	Port number of the remote host.

RETURN VALUE

- ≥0: Success, number of bytes sent.
- 1: Failure.
- 2: Failed because hardware address not resolved.

LIBRARY

UDP.LIB

SEE ALSO

[udp_send](#), [udp_xsendto](#), [udp_recv](#), [udp_recvfrom](#), [udp_open](#)

udp_waitopen

```
int udp_waitopen( udp_Socket *s, int iface, word lport, longword
    remip, word port, dataHandler_t datahandler, long buffer,
    int buflen, longword millisecs );
```

DESCRIPTION

This function is identical to `udp_extopen()`, except that it waits a specified amount of time for the hardware address of the destination to be resolved.

While waiting, this function calls `tcp_tick()`.

PARAMETERS

s	Pointer to socket.
iface	Local interface on which to open the socket. This parameter is supported as of Dynamic C 7.30. With earlier version of DC, this parameter should be <code>IF_DEFAULT</code> .
lport	Local port.
remip	Acceptable remote ip, or 0 for all.
port	Acceptable remote port, or 0 for all.
datahandler	Function to call when data is received, <code>NULL</code> for placing data in the sockets receive buffer.
buffer	Address of user-supplied socket buffer in <code>xmem</code> , 0 to use a buffer from the socket buffer pool.
buflen	Length of user-supplied socket buffer.
millisecs	Maximum milliseconds to wait for the hardware address to be resolved.

RETURN VALUE

>0: Successfully opened socket.

0: Timed out without resolving address.

-1: Error opening socket (e.g., buffer could not be allocated).

LIBRARY

`UDP.LIB`

SEE ALSO

[udp_extopen](#), [sock_resolved](#)

udp_waitsend

```
int udp_waitsend( udp_socket *s, char *buffer, int len,  
                 longword remip, word remport, word millisecs );
```

DESCRIPTION

This is identical to `udp_sendto()`, except that it will block for up to the specified amount of time waiting for the hardware address to be resolved. Normally, you should not have to specify more than 100 ms for the time out. If it takes longer than this, the destination is probably unavailable.

PARAMETERS

s	UDP socket on which to send the datagram.
buffer	Buffer that contains the UDP datagram.
len	Length of the UDP datagram.
remip	IP address of the remote host.
remport	Port number of the remote host.
millisecs	Number of milliseconds to wait for hardware address resolution. Reasonable values are between 50 and 750 milliseconds.

RETURN VALUE

≥0: Number of bytes sent.
-1: Failure (invalid UDP socket etc.).
-2: Failure (timed out, no datagram sent).

LIBRARY

UDP.LIB

SEE ALSO

[udp_sendto](#), [udp_recvfrom](#), [udp_bypass_arp](#)

udp_xsendto

```
int udp_xsendto( udp_socket *s, long buffer, int len, longword remip,  
                word remport );
```

DESCRIPTION

Send a single UDP datagram on a UDP socket. It will send the datagram to the IP address specified by `remip`, and the port specified by `remport`. Note that this function can be used even on a socket that has been "connected" to a remote host and port.

This function is identical to `udp_sendto()` except that the data address is specified as a physical address.

PARAMETERS

s	UDP socket on which to send the datagram.
buffer	Buffer that contains the UDP datagram.
len	Length of the UDP datagram.
remip	IP address of the remote host.
remport	Port number of the remote host.

RETURN VALUE

≥0: Number of bytes sent.
-1: Failure.
-2: Failure (hardware address not resolved).

LIBRARY

UDP.LIB

SEE ALSO

[udp_send](#), [udp_recv](#), [udp_recvfrom](#), [udp_open](#), [udp_sendto](#)

virtual_eth

```
int virtual_eth( word real_iface, longword ipaddr, longword
    netmask, void * resv );
```

DESCRIPTION

Create a new virtual ethernet interface. You must #define VIRTUAL_ETH to a positive number (1-6) for this function to work. The macro VIRTUAL_ETH gives the maximum number of virtual interfaces.

Virtual ethernet interfaces have some restrictions:

- You cannot use DHCP.
- Broadcast/multicast packets are not received.
- Some `ifconfig()` settings (such as MTU size) are not settable.
- Once a virtual interface is created, it cannot be destroyed. In practice, this means that all virtual interfaces should be created at boot time (after `sock_init()`).

The virtual interface will be created in the same up/down state as the real interface. Changes to the up/down state of the real interface will affect all virtual interfaces tied to that interface.

The callback function for a virtual interface is set to NULL.

PARAMETERS

real_iface	The real interface to use. This must be <code>IF_ETH0</code> , or may be <code>IF_ETH1</code> for boards with two Ethernet chips.
ipaddr	The IP address to assign this interface. This must not be the same as the IP address of any other interface.
netmask	Netmask to use. If zero, then the netmask of the real interface will be used.
resv	Pointer reserved for future use. Pass as NULL.

RETURN VALUE

-1: Failed because VIRTUAL_ETH was not defined, or the number of virtual interfaces exceeds the value specified by VIRTUAL_ETH, or the `real_iface` parameter was not valid.

Otherwise: returns the interface number to use for this virtual interface. This should be passed to any other function that requires the interface number to be specified.

LIBRARY

NET.LIB

SEE ALSO

[ifconfig](#)

wifi_ssid_to_str

```
far char * wifi_ssid_to_str (char far *dest, char far *ssid, int len);
```

DESCRIPTION

This function creates a null-terminated string of printable characters from a given SSID. Since the SSID can contain any byte (including nulls and characters > 0x7F), you can use `wifi_ssid_to_str()` to create a printf-safe string.

PARAMETERS

dest	Pointer to a 33-byte buffer to receive the printable string.
ssid	Pointer to a 0 to 32-byte SSID.
len	Length of SSID.

RETURN VALUE

Returns pointer to user-supplied 33-byte buffer (`dest`) containing printf-safe string.

NOTE:

To perform the conversion in place (re-use the buffer):

```
int i;
char b[33];
ifconfig (IF_WIFIO, IFG_WIFI_SSID, &i, b, IFS_END);
printf ("SSID: [%ls]\n", wifi_ssid_to_str (b, b, i));
```

LIBRARY

WIFI_WLN_API.LIB

SEE ALSO

[ifconfig](#) (IFG_WIFI_SSID option)

Index

Numerics

2MSL 266
 3-way handshake 57

A

ARP_CONFLICT_CALLBACK 90
 ARP_LONG_EXPIRY 89
 ARP_NO_ANNOUNCE 90
 ARP_PERSISTENCE 90
 ARP_PURGE_TIME 89
 ARP_ROUTER_TABLE_SIZE 41, 90
 ARP_SHORT_EXPIRY 89
 ARP_TABLE_SIZE 41, 90

B

bandwidth 77, 93
 BOOTP/DHCP
 _bootpdata 39
 _bootpdone 38
 _bootperror 39
 _bootphost 38
 _bootpon 37
 _bootpsize 38
 _bootptimeout 38
 _dhcpghost 38
 _dhcplife 38
 _dhcpt1 38
 _dhcpt2 38
 _smtpsrv 39
 _survivebootp 37
 broadcast packets 53, 61, 63, 270, 272, 274, 277
 buffer sizes 56

C

callbacks
 CGI 88
 interface status 15
 IP address conflict 90
 TCP and UDP data handlers 69
 checksums 225
 communication channel 77

D

daemons
 tcp_tick 267
 data handler callbacks 69
 DCRTCP_DEBUG 45
 DCRTCP_VERBOSE 45
 DHCP_CHECK 36
 DHCP_CLASS_ID 37
 DHCP_CLIENT_ID 37
 DHCP_CLIENT_ID_LEN 37
 DHCP_CLIENT_ID_MAC 37
 DHCP_USE_BOOTP 36
 DHCP_USE_TFTP 37
 DISABLE_DNS 35, 91
 DISABLE_TCP 35
 DNS 91
 DNS_MAX_DATAGRAM_SIZE 41, 91
 DNS_MAX_NAME 41, 91
 DNS_MAX_RESOLVES 41, 91
 DNS_MIN_KEEP_COMPLETED 45, 92
 DNS_NUMBER_RETRIES 45, 92
 DNS_RETRY_TIMEOUT 45, 91
 DNS SOCK_BUF_SIZE 42, 92
 drivers
 link layer 6

E

ephemeral connection 54
 error messages 227
 ETH_MAXBUFS 41
 ETH_MTU 40
 Ethernet
 ports 3
 Ethernet Transmission Unit 216

F

Function Reference
 Addressing
 _arp_resolve 117
 _arp_send_gratuitous 122
 arp_getArpData 116
 arp_getArpGateData 116
 arpcache_create 108
 arpcache_flush 109
 arpcache_hwa 110

arpcache_ipaddr	112
arpcache_load	113
arpcache_search	115
arpresolve_check	118
arpresolve_ipaddr	119
arpresolve_start	120
arpresolve_start_iface	121
dhcp_acquire	125
dhcp_get_timezone	126
dhcp_release	127
getdomainname	128
gethostid	129
gethostname	130
getpeername	131
getsockname	132
psocket	186
resolve	187
resolve_cancel	188
resolve_name_check	189
resolve_name_start	190
router_add	192
router_del_all	193
router_delete	193
router_for	194
router_for_iface	195
router_print	197
router_printall	198
setdomainname	200
sethostid	201
sethostname	202
udp_bypass_arp	268
Configuration	
ifconfig	135
tcp_config	258
wifi_ioctl	26
Data Conversion	
atoh	123
htohl	133
htons	134
inet_addr	163
inet_ntoa	164
ntohl	177
ntohs	178
rip	191
Ethernet	
pd_getaddress	179
pd_havelink	180
pd_powerdown	181
pd_powerup	182
Initialization	
sock_init	222
sock_init_or_exit	223
sock_tick	247
Interface	
ifdown	158
ifpending	159
ifstatus	160
ifup	161
ip_iface	165
ip_print_ifs	166
is_valid_iface	169
sock_iface	221
virtual_eth	282
Modem	
ModemClose	170
ModemConnected	170
ModemExpect	171
ModemHangup	171
ModemInit	172
ModemOpen	172
ModemReady	173
ModemRinging	173
ModemSend	174
ModemStartPPP	174
Multicast	
multicast_joingroup	175
multicast_leavegroup	176
Ping	
_chk_ping	124
_ping	183
_send_ping	199
PPP	
PPPActive	184
PPPnegotiateIP	184
PPPsetAuthenticatee	185
PPPsetAuthenticator	185
PPPshutdown	186
Socket Configuration	
sock_mode	224
sock_set_tos	241
sock_set_ttl	242
tcp_clearreserve	257
tcp_reserveport	266
Socket Connection	
_abort_socks	107
sock_abort	203
sock_close	210
sock_established	214
sock_waiting	250
tcp_keepalive	261
Socket I/O Buffer	
sock_rleft	231
sock_rbsize	232
sock_rbused	233
sock_tleft	244
sock_tbsize	245
sock_tbused	246

Socket Status	
ip_timer_expired	167
ip_timer_init	168
sock_alive	204
sock_bytesready	209
sock_dataready	211
sock_error	213
sock_perror	227
sock_readable	235
sock_resolved	240
sock_writable	252
sockerr	212
sockstate	243
tcp_tick	267
TCP Socket I/O	
sock_aread	205
sock_awrite	206
sock_axread	207
sock_axwrite	208
sock_fastread	215
sock_fastwrite	216
sock_flush	217
sock_flushnext	218
sock_getc	219
sock_gets	220
sock_preread	228
sock_putc	229
sock_puts	230
sock_read	234
sock_write	253
sock_xfastread	254
sock_xfastwrite	255
sock_yield	256
tcp_extlisten	259
tcp_extopen	260
tcp_listen	262
tcp_open	264
TCP/IP Stack	
sock_init	222
sock_init_or_exit	223
tcp_tick	267
UDP Socket I/O	
udp_close	269
udp_extopen	270
udp_open	272
udp_peek	274
udp_recv	275
udp_recvfrom	276
udp_send	277
udp_sendto	278
udp_waitopen	279
udp_waitsend	280
udp_xsendto	281
UDP Socket I/O (pre-DC 7.05)	
sock_fastread	215
sock_fastwrite	216
sock_read	234
sock_recv	236
sock_recv_from	238
sock_recv_init	239
sock_write	253
udp_close	269
udp_open	272
Wi-Fi	
wifi_ioctl	26
wifi_ssid_to_str	283
H	
host group	93
I	
ICMP_TOS	47
IF_*	4
IFCONFIG_*	43
IGMP	93
interfaces	
configuration	8–14
enable/disable support	5
single	7
sum of physical	6
supported types	3
IP addresses	
broadcast packets	61, 63
default	9, 42
directed ping	13
dynamic configuration	11
last-used DHCP server	38
last-usedBOOTP/TFTP server	38
lease	11, 38
mail server	39
origin of received datagram	64
runtime configuration	12
setting to zero	56
sources of	9
Zconsole configuration	14
ISPs and MAC addresses	14
K	
KEEPALIVE_NUMRETRYS	44
KEEPALIVE_WAITTIME	44
L	
latency	78, 88
link layer drivers	6

M

MAC address	14, 85
macros	
ARP	89
BOOTP/DHCP	36
buffer/resource sizing	39
DNS	91
including additional functionality	36
interface configuration	4
interface configuration (7.30 and later)	42
interface selection	5
link layer driver	7
miscellaneous	46
network configuration (pre 7.30)	42
program debugging	45
removing unwanted functionality	35
timers and counters	44
TOS and TTL	46
MAX_COOKIES	41
MAX_DOMAIN_LENGTH	42
MAX_NAMESERVERS	41
MAX_RESERVEPORTS	41
MAX_SOCKET_LOCKS	39, 73
MAX_SOCKETS	39
MAX_STRING	41
MAX_TCP_SOCKET_BUFFERS	39, 260
MAX_UDP_SOCKET_BUFFERS	40
memmap	65
modem library	96
MSS (maximum segment size)	40
MTU	216
multicasting	93, 270
multitasking	73
MY_DOMAIN	40, 42
MY_GATEWAY	42
MY_IP_ADDRESS	42
MY_NAMESERVER	42
MY_NETMASK	42

N

Nagle algorithm	80, 224
NET_ADD_ENTROPY	46
NET_COARSELOCK	46
network addressing	89

O

optimizations	77
---------------------	----

P

packet	
acknowledgement	78, 80
processing	66
size	79

TOS	85
performance optimizing	77
PKTDRV	7
port numbers	54
PPP driver	95
PPP_MTU	40

R

RETRAN_STRAT_TIME	44, 82
router	89, 90, 94
RTT	78

S

SOCK_BUF_SIZE	40
socket	
abort all	107
buffers	55
data structure	54
default mode	60
definition	54
empty line vs empty buffer	209
locks	73, 224
stack	
configuration	3–8
initialization	7

T

TCP socket	53
active open	57
control functions	58
I/O functions	60
blocking	69
non-blocking	68
listen queue	58
passive open	56
TCP/IP	
initialization	66
skeleton program	65
TCP_BUF_SIZE	40
TCP_CONNTIMEOUT	44
TCP_FASTSOCKETS	46
TCP_LAZYUPD	45, 83
tcp_MaxBufSize	40
TCP_MAXPENDING	41
TCP_MAXRTO	44
TCP_MINRTO	45, 82
TCP_NO_CLOSE_ON_LAST_READ	46
TCP_OPENTIMEOUT	44
TCP_STATS	45
TCP_SYNQTIMEOUT	44
TCP_TOS	47
TCP_TTL	47
TCP_TWTIMEOUT	44, 83

TCPCONFIG	9, 42
throughput	77, 88
tick rates	66

U

UDP	
broadcast packets	61
performance	61
UDP socket	
checksum	61
functions	61
open and close	63
read	64
write	63
UDP_BUF_SIZE	40
UDP_TOS	47
UDP_TTL	47
USE_DHCP	36
USE_ETHERNET	5, 42
USE_PPOE	5
USE_PPP_SERIAL	5, 43, 99
USE_PPPOE	43, 98
USE_RESERVEDPORTS	58
USE_SNMP	36
USE_WIFI	5

W

Wi-Fi	16
-------------	----

Dynamic C TCP/IP Functions

Listed Alphabetically

Symbols		
_abort_socks	107
_arp_send_gratuitous	122
_chk_ping	124
_ping	183
_send_ping	199
A		
arp_getArpData	116
arp_getArpGateData	116
arp_resolve	117
arpcache_create	108
arpcache_flush	109
arpcache_hwa	110
arpcache_iface	111
arpcache_ipaddr	112
arpcache_load	113
arpcache_search	115
arpresolve_check	118
arpresolve_ipaddr	119
arpresolve_start	120
arpresolve_start_iface	121
atons	123
D		
dhcp_acquire	125
dhcp_get_timezone	126
dhcp_release	127
G		
getdomainname	128
gethostid	129
gethostname	130
getpeername	131
getsockname	132
H		
htonl	133
htons	134
I		
ifconfig	135
ifdown	158
ifpending	159
ifstatus	160
ifup	161
inet_addr	163
inet_ntoa	164
ip_iface	165
ip_print_ifs	166
ip_timer_expired	167
ip_timer_init	168
is_valid_iface	169
M		
ModemClose	170
ModemConnected	170
ModemExpect	171
ModemHangup	171
ModemInit	172
ModemOpen	172
ModemReady	173
ModemRinging	173
ModemSend	174
ModemStartPPP	174
multicast_joingroup	175
multicast_leavegroup	176
N		
ntohl	177
ntohs	178
P		
pd_getaddress	179
pd_havelink	180
pd_powerdown	181
pd_powerup	182
PPPactive	184
PPPnegotiateIP	184
PPPsetAuthenticatee	185
PPPsetAuthenticator	185
PPPshutdown	186
psocket	186

R

resolve	187
resolve_cancel	188
resolve_name_check	189
resolve_name_start	190
rip	191
router_add	192
router_del_all	193
router_delete	193
router_for	194
router_for_iface	195
router_print	197
router_printall	198

S

setdomainname	200
sethostid	201
sethostname	202
sock_abort	203
sock_alive	204
sock_aread	205
sock_awrite	206
sock_axread	207
sock_axwrite	208
sock_bytesready	209
sock_close	210
sock_dataready	211
sock_error	213
sock_established	214
sock_fastread	215
sock_fastwrite	216
sock_flush	217
sock_flushnext	218
sock_getc	219
sock_gets	220
sock_iface	221
sock_init	222
sock_init_or_exit	223
sock_mode	224
sock_noflush	226
sock_perror	227
sock_preread	228
sock_putc	229
sock_puts	230
sock_rbleft	231
sock_rbsize	232
sock_rbused	233
sock_read	234

sock_readable	235
sock_recv	236
sock_recv_from	238
sock_recv_init	239
sock_resolved	240
sock_set_tos	241
sock_set_ttl	242
sock_tbleft	244
sock_tbsize	245
sock_tbused	246
sock_tick	247
sock_wait_closed	248
sock_wait_established	249
sock_wait_input	251
sock_waiting	250
sock_writable	252
sock_write	253
sock_xfastread	254
sock_xfastwrite	255
sock_yield	256
sockerr	212
sockstate	243

T

tcp_clearreserve	257
tcp_config	258
tcp_extlisten	259
tcp_extopen	260
tcp_keepalive	261
tcp_listen	262
tcp_open	264
tcp_reserveport	266
tcp_tick	267

U

udp_bypass_arp	268
udp_close	269
udp_extopen	270
udp_open	272
udp_peek	274
udp_recv	275
udp_recvfrom	276
udp_send	277
udp_sendto	278
udp_waitopen	279
udp_waitsend	280
udp_xsendto	281

V

virtual_eth	282
-------------------	-----

W
wifi_ssid_to_str283

Dynamic C TCP/IP Functions

Listed by Category

Addressing

_arp_resolve	117
_arp_send_gratuitous	122
arp_getArpData	116
arp_getArpGateData	116
arpcache_create	108
arpcache_flush	109
arpcache_hwa	110
arpcache_iface	111
arpcache_ipaddr	112
arpcache_load	113
arpcache_search	115
arpresolve_check	118
arpresolve_ipaddr	119
arpresolve_start	120
arpresolve_start_iface	121
dhcp_acquire	125
dhcp_get_timezone	126
dhcp_release	127
getdomainname	128
gethostid	129
gethostname	130
getpeername	131
getsockname	132
psocket	186
resolve	187
resolve_cancel	188
resolve_name_check	189
resolve_name_start	190
router_add	192
router_del_all	193

router_delete	193
router_for	194
router_for_iface	195
router_print	197
router_printall	198
setdomainname	200
sethostid	201
sethostname	202
udp_bypass_arp	268

Configuration

ifconfig	135
tcp_config	258

Data Conversion

aton	123
htonl	133
htons	134
inet_addr	163
inet_ntoa	164
ntohl	177
ntohs	178
rip	191

Ethernet

pd_getaddress	179
pd_havelink	180
pd_powerdown	181
pd_powerup	182

Initialization

sock_init	222
sock_init_or_exit	223

sock_tick	247
Interface	
ifdown	158
ifpending	159
ifstatus	160
ifup	161
ip_iface	165
ip_print_ifs	166
is_valid_iface	169
sock_iface	221
virtual_eth	282
Modem	
ModemClose	170
ModemConnected	170
ModemExpect	171
ModemHangup	171
ModemInit	172
ModemOpen	172
ModemReady	173
ModemRinging	173
ModemSend	174
ModemStartPPP	174
Multicast	
multicast_joingroup	175
multicast_leavegroup	176
Ping	
_chk_ping	124
_ping	183
_send_ping	199
PPP	
PPPactive	184
PPPnegotiateIP	184
PPPsetAuthenticatee	185
PPPsetAuthenticator	185
PPPshutdown	186

Socket Configuration

sock_mode	224
sock_set_tos	241
sock_set_ttl	242
tcp_clearreserve	257
tcp_reserveport	266

Socket Connection

_abort_socks	107
sock_abort	203
sock_close	210
sock_established	214
sock_waiting	250
tcp_keepalive	261

Socket I/O Buffer

sock_rbleft	231
sock_rbsize	232
sock_rbused	233
sock_tbleft	244
sock_tbsize	245
sock_tbused	246

Socket Status

ip_timer_expired	167
ip_timer_init	168
sock_alive	204
sock_bytesready	209
sock_dataready	211
sock_error	213
sock_perror	227
sock_readable	235
sock_resolved	240
sock_writable	252
sockerr	212
sockstate	243
tcp_tick	267

TCP Socket I/O

sock_aread	205
------------------	-----

sock_awrite	206
sock_axread	207
sock_axwrite	208
sock_fastread	215
sock_fastwrite	216
sock_flush	217
sock_flushnext	218
sock_getc	219
sock_gets	220
sock_noflush	226
sock_preread	228
sock_putc	229
sock_puts	230
sock_read	234
sock_write	253
sock_xfastread	254
sock_xfastwrite	255
sock_yield	256
tcp_extlisten	259
tcp_extopen	260
tcp_listen	262
tcp_open	264

sock_recv_init	239
----------------------	-----

Wi-Fi

wifi_ssid_to_str	283
------------------------	-----

UDP Socket I/O

udp_close	269
udp_extopen	270
udp_open	272
udp_peek	274
udp_recv	275
udp_recvfrom	276
udp_send	277
udp_sendto	278
udp_waitopen	279
udp_waitsend	280
udp_xsendto	281

UDP Socket I/O (pre-DC 7.05)

sock_recv	236
sock_recv_from	238

