Interfacing MicroC/OS to TCP/IP

Interfacing Small Systems to TCP/IP

- TCP/IP is a complicated protocol that requires substantial memory space and program complexity.

- Special care is required when interfacing small systems, with limited memory and CPU resources, to TCP/IP.

- As implementation examples, we will consider how MicroC/OS has been interfaced to two TCP/IP stacks:
  1. NetBurner's TCP/IP stack (the ECE 315 lab system)
  2. The Lightweight IP (lwIP) portable, open-source TCP/IP stack. [Adam Dunkels, “Design and Implementation of the LWIP TCP/IP Stack,” Swedish Institute of Computer Science, Feb. 20, 2001.]

Opportunities for Saving Time and Space

- Strict partitioning of the TCP/IP interface according to the protocol layers can be inefficient.

  Ex. It may be more efficient at times for TCP to be able to look directly into the IP header without always having to go through the IP layer software.

- We want to avoid unnecessary copying of data and packet buffers between the application and the communication subsystem, or between different layers in the protocol stack.

  Ex. We may want the application to use the same buffers as the communication subsystem even if this violates strict partitioning between applications and the TCP/IP stack and operating system.

Implement the Protocol Layers as Tasks?

- One could employ separate tasks for each protocol layer:
  (4) Application task(s)
  (3) Transport (TCP) layer task
  (2) Internetworking (IP) layer task
  (1) Device driver task (for Ethernet, RS-232C, etc.)
  This approach was taken by NetBurner’s TCP/IP stack.
- *Advantages*:
  – interlayer interfaces strictly defined and enforceable
  – debugging and code maintenance is easier
  – can more easily modify protocol settings at run-time
- *Disadvantages*:
  – larger required number of task context switches
  – need to carefully control access to shared buffers
  – access to multi-layer protocol data is slowed down

Some Other Implementation Options

- Embed the TCP/IP stack inside the kernel. User tasks must access the stack using system calls (also called “traps”).

  Advantages: Fast & efficient; users shielded from stack details.
  Disadvantages: More difficult to change or modify the stack.

- Combine protocol layers into one task, outside the kernel. User code may, in some cases, be combined with this same task.

  Advantages: Protocol stack is more portable across kernels. Opportunities to gain efficiency in space & time.
  Disadvantages: Stack software is less clearly partitioned apart from user code.

- Lightweight IP takes the last approach, with the option of keeping user code in the same or a separate task from the stack task.

The NetBurner TCP/IP Stack in MicroC/OS

- NetBurner's TCP/IP stack is partitioned across several tasks that handle the different protocol layers:

  Priority 38: Ethernet hardware (FEC) driver task

  Priority 39: IP entity task

  Priority 40: TCP entity task

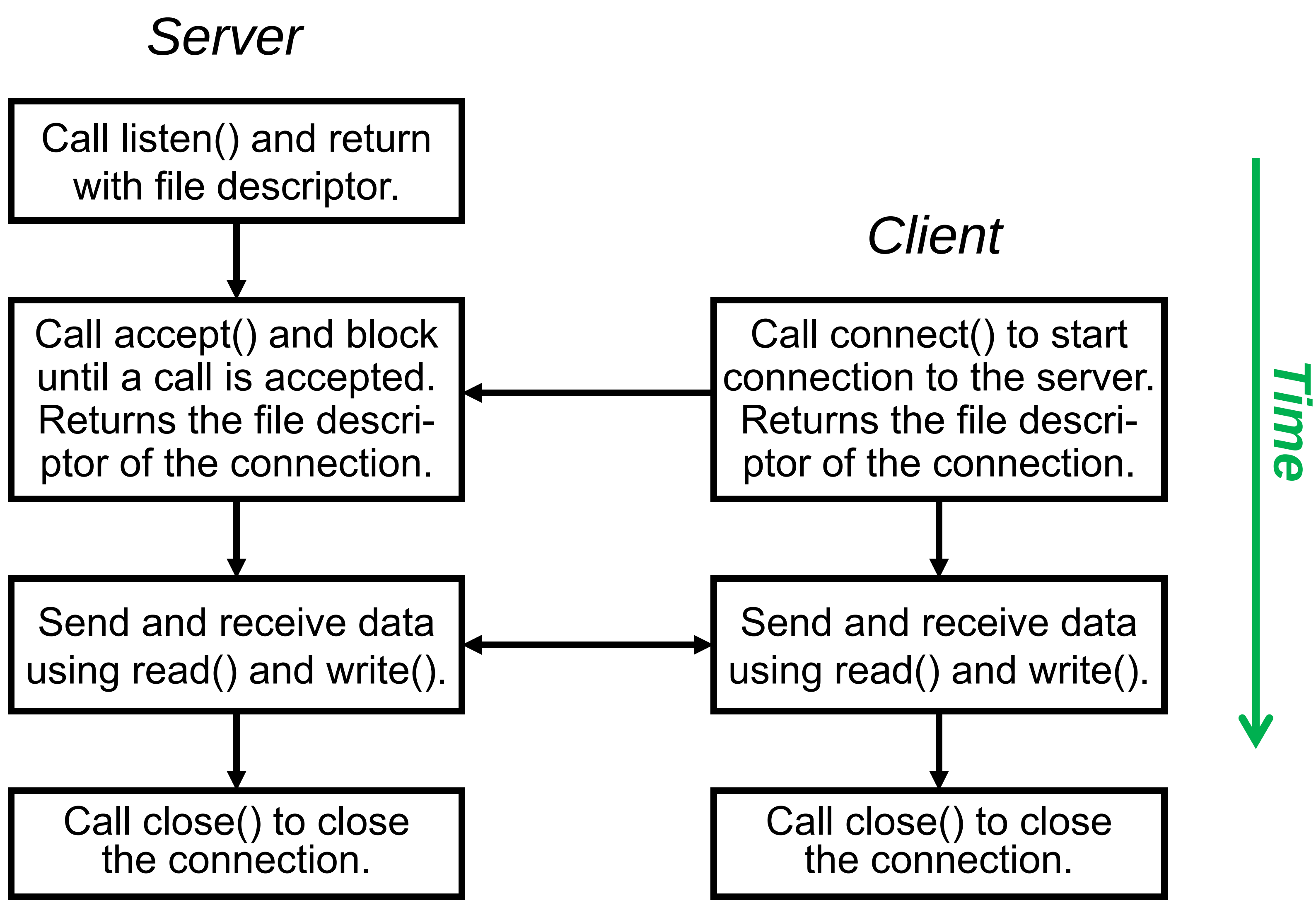
  Priority 44: PPP entity task (for TCP/IP over RS-232C)

  Priority 45: HTTP web server task (created by StartHTTP)

  Priority 50: Default priority of the UserMain() task

- The FEC driver task has the highest priority (i.e., lowest priority number in MicroC/OS) so that Ethernet hardware actions will be handled promptly before IP and TCP.

NetBurner's Client-Server Architecture



TCP/IP Application Programming Interface (1)

New data types introduced in the TCP/IP stack:

- IPADDR - data structure that holds one IP address or mask

File descriptor error return codes (all negative values):

- TCP_ERR_TIMEOUT - connection timed out
- TCP_ERR_NOCON - connection not yet established
- TCP_ERR_CLOSING - connection closed and not available
- TCP_ERR_NOSUCH_SOCKET - socket does not exist
- TCP_ERR_NONE_AVAIL - no more sockets are available
- TCP_ERR_CON_RESET - connection reset by other side
- TCP_ERR_CON_ABORT - internal stack error

TCP/IP Application Programming Interface (2)

```
void InitializeStack( IPADDR IP_addr = 0,
                      IPADDR IP_mask = 0,
                      IPADDR IP_gateway = 0 );
```

- function that initializes the TCP/IP stack
- typically called by UserMain()
- "IP_addr" is the IP address of the local node
- "IP_mask" is the IP mask of the local node
- "IP_gateway is the IP address of the gateway for the local node
- If called with no inputs, then default IP values are copied from the system configuration record.

```
void KillStack( );
```

- function that shuts down the TCP/IP stack
- typically called by UserMain()

TCP/IP Application Programming Interface (3)

int listen(IPADDR IP_addr, WORD port_num, BYTE max_pends);

- called by the server to put the TCP entity into the LISTEN state
- "IP_addr" specifies a client IP; value INADDR_ANY means that a connection/call will be accepted from any client IP
- "port_num" specifies the listening port number on the server
- "max_pends" specifies maximum number of pending connections
- return value is the listening file descriptor, i.e. a dummy socket

int accept(int fdl, IPADDR *Pclient_IP, WORD *Pclient_port,
WORD timeout);

- called by the server to block until a client call is received
- "fdl" is the listening file descriptor, i.e. the dummy listening socket
- "Pclient_IP" points to the IP address of a new calling client
- "Pclient_port" points to the port number on the client
- Pclient_IP and Pclient_port can be left null if information not wanted
- "timeout" is the maximum number of ticks waiting for a call;
  a "timeout" value of 0 means wait forever for the next call
- return value is the network file descriptor, i.e. the working socket

TCP/IP Application Programming Interface (4)

int read(int fdnet, char *buf, int buf_siz);

- used by servers & clients to receive a char array payload
- “fdnet” is the network file descriptor, i.e. the working socket
- “buf” stores the retrieved payload, which is a char array
- “buf_siz” is the maximum number of chars to be read
- positive return value is the actual number of new chars in "buf"
- negative return code means the socket was closed by the other side.

int write(int fdnet, char *buf, buf_siz);

- used by servers & clients to send a char array payload
- “fdnet” is the network file descriptor, i.e. the working socket to write
- “buf” holds the payload to be written, which is a char array
- “buf_siz” is the number of chars to be written
- return code is the number of chars actually written

TCP/IP Application Programming Interface (5)

int ReadWithTimeout(int fdnet, char *buf, int buf_siz,
unsigned long timeout);

- used by servers & clients to receive a char array payload
- “fdnet” is the network file descriptor, i.e. the working socket
- “buf” stores the received payload, which is a char array
- “buf_siz” is the maximum number of chars to be read
- "timeout" is a timeout to socket closure, expressed in timer ticks
- a positive return value is the number of chars in char_buf
- a negative return value means the other party closed the socket

void writestring(int fdnet, char *buf);

- an alternative to the write() function
- "fdnet" is the network file descriptor, i.e. the working socket
- writes a null-terminated string to a network file descriptor

TCP/IP Application Programming Interface (6)

int connect(IPADDR remote_IP, WORD local_port, WORD remote_port, DWORD timeout);

- connects a client to the specified port on a remote server
- "remote_IP" specifies the IP address of the remote server
- "local_port" specifies the port number to use for the connection; a value of 0 causes the stack to choose an unused port
- "remote_port" specifies the port number on the server
- "timeout" is a timeout, expressed in timer ticks; a value of 0 means that the function will wait forever
- return value is the network file descriptor, i.e. the working socket

void close(fdnet);

- called by both servers & clients to end a TCP connection
- "fdnet" is the network file descriptor, i.e. the working socket

Example: A Simple Echo Server (1)

```
#include “predef.h”
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <ip.h>
#include <tcp.h>

#define ECHO_LISTEN_PORT  23    // use the TELNET port
#define RX_BUFSIZE  4096

DWORD EchoServerTaskStack[USER_TASK_STK_SIZE];
                          __attribute__(( aligned(4)));

const char *Echo_server = "Simple echo server";

char RXBuffer[RX_BUFSIZE];
```

Example: A Simple Echo Server (2)

```
void EchoServerTask( void *pd )
{
   IPADDR client_IP;
   WORD port;
   int ListenPort = *(int *)pd;
   int fdListen = listen(INADDR_ANY, ListenPort, 7);
   if (fdListen > 0) {
      while (1) {
         int fdnet = accept(fdListen, &client_IP, &port, 0);
         while (fdnet > 0) {
            writestring( fdnet, "Welcome to the server\r\n" );
            int n = 0;
            do {
               n = read( fdnet, RXBuffer, RX_BUFSIZE );
               write( fdnet, RXBuffer, n );
            } while ((n > 0) && (RXBuffer[0] != '.'))
            close( fdnet );
            fdnet = 0;
         } // end while (fdnet > 0)
      } // end while (1)
   } // if (fdListen > 0)
}
```

Example: A Simple Echo Server (3)

```
void UserMain( void *pd)
{
   InitializeStack();

   if (EthernetIP == 0) { GetDHCPAddress(); }
   EnableAutoUpdate();

   OSChangePrio(MAIN_PRIO);

   OSTaskCreate( EchoServerTask,
       (void *) ECHO_LISTEN_PORT,
       (void *) &EchoServerTaskStack[USER_TASK_STK_SIZE],
       (void *) EchoServerTaskStack,
       MAINPRIO-1) ;  // higher priority than UserMain()

   while (1) {
      OSTimeDly( TICKS_PER_SECOND * 5 );
   } // end while (1)
}
```

NetBurner's HTTP Server (1)

- The simple echo server could be used to build an HTTP server; however, NetBurner already provides an extensible HTTP server.
- NetBurner’s HTTP server (a “webserver”) is implemented as a task at priority HTTP_PRIO, which is 45 by default.
- Required steps to use NetBurner’s webserver:
  - include the header file “http.h”
  - include the header file “htmlfiles.h”
  - initialize the TCP/IP stack by calling “InitializeStack()”
  - start up the HTTP server by calling “StartHTTP()”
- A default web page will be returned for GET requests to port 80.
- The HTTP server is shut down by calling “StopHTTP()”.

NetBurner's HTTP Server (2)

New types that are defined and used in the HTTP server:

```
typedef  char *          PSTR      // pointer to a string
typedef  const char *  PCSTR     // pointer to a constant string
```

void StartHTTP(WORD port = 80);

- starts up the HTTP server task
- default HTTP port of 80 can be overridden
- usually called by UserMain();
- must have previously called InitializeStack();

void StopHTTP();

- shuts down the HTTP server task

NetBurner's HTTP Server (3)

```
// Default simple handler of HTTP GET requests
int BaseDoGet( int sock, PSTR url, PSTR rxBuffer )
{
   if (*url == 0) {   // null url, so GET default webpage
      RedirectResponse( sock, url_of_default_webpage );
      // url_of_default_webpage is typically "HTML/index.htm“
      // All other webserver .htm files are in the HTML dir.
      return 1;
   }
   if ( httpstricmp(url,"ECHO") ) {  // ECHO server loop
      while (*url==0) url++;         // Advance past nul's
      SendTextHeader( sock );        // Send HTML header
      *url = ' ';                    // Insert a space char
      writestring( sock, rxBuffer ); // Echo back the message
      return 1;
   }
   if (!SendFullResponse( url, sock )) { // GET webpage
      NotFoundResponse( sock, url );  // else 404 not found
   }
   return 0;
}
```

NetBurner's HTTP Server (4)

void RedirectResponse(int fd, PCSTR url);

- redirect the current GET request to the webpage at "url"
- send the webpage to socket "fd"

int SendFullResponse(PCSTR url, int fd);

- responding by GET-ing the stored webpage for the given "url"
- the webpage must have been stored previously in the html subdirectory of the project directory before building the project
- the webpage may have dynamic parts that change at runtime
- send the HTTP header plus webpage back to socket "fd"
- returns 1 if webpage was found; otherwise, returns 0

void NotFoundResponse(int fd, PCSTR url);

- no webpage was found on the server with the given "url"
- send back an HTTP "not found" response to socket "fd"

NetBurner's HTTP Server (5)

int httpstricmp(PCSTR string, PCSTR pattern);

- "string" contains an array of chars
- "pattern" contains an array of UPPER CASE chars
- returns 1 if the shorter "string" or "pattern" appears as a prefix of the longer array of "string" or "pattern"; otherwise, returns 0

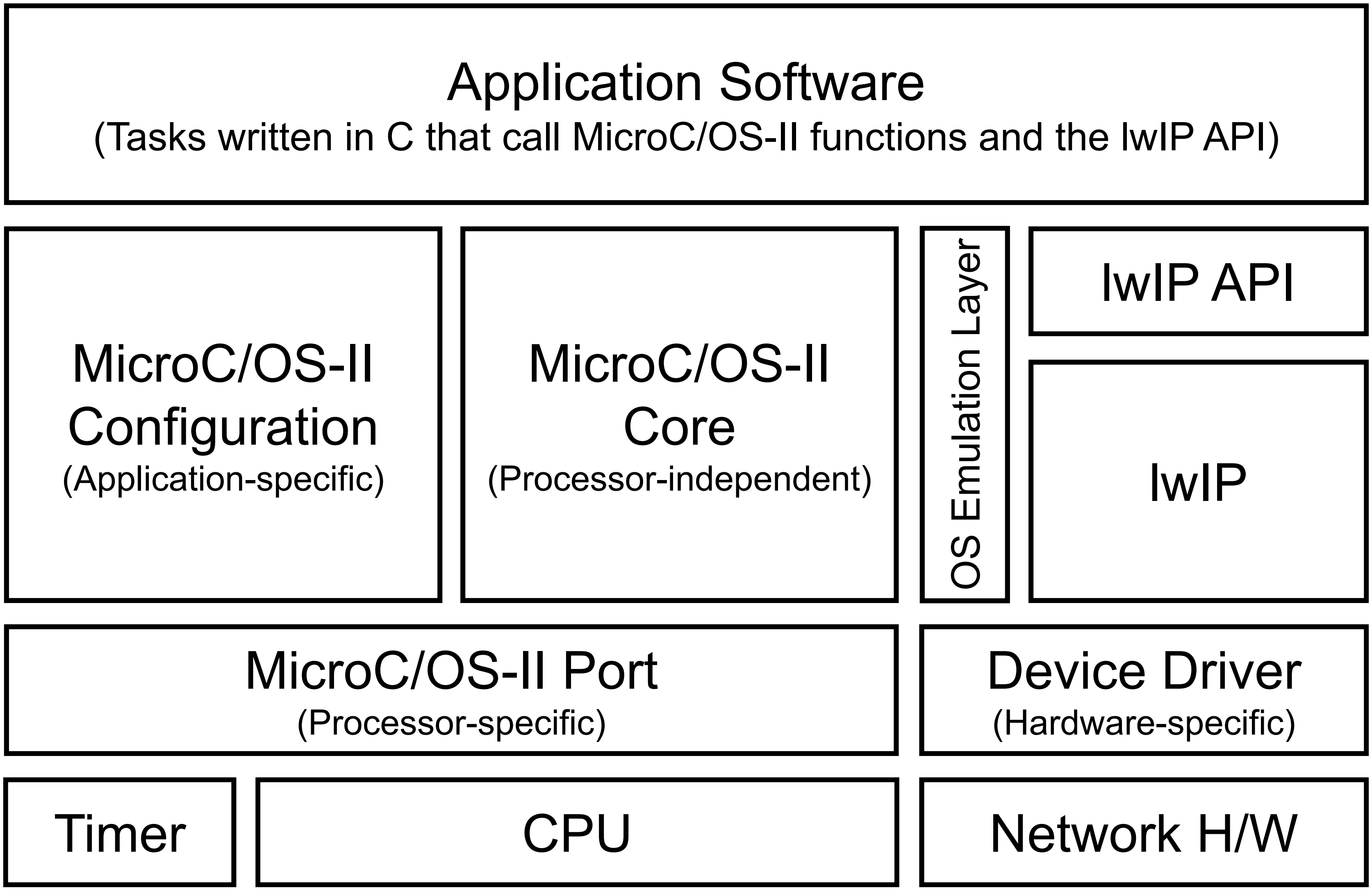
void SendTextHeader(int fd);

- send the appropriate HTTP response header for a text reply
- must be the first part of the HTTP response to socket "fd"
- followed by a write of a plain text file
- used to response to HTTP ECHO requests

void SendHTMLHeader(int fd);

- send the appropriate HTTP response header for an HTML file
- followed by a write of an HTML file

Architecture showing MicroC/OS-II with lwIP



Components of Lightweight IP

lwIP Application Program Interface (API):

- simplified function calls for accessing TCP/IP network
- very similar to the socket API in Unix

lwIP Process:

- kernel/OS-independent implementation of TCP/IP

Operating System Emulation Layer:

- minimal set of functions required of the O.S. by lwIP

Network Hardware Interface:

- provides access to physical layer: Ethernet, RS-232C, Bluetooth, wireless LAN (IEEE 802.11b, 802.11a), etc.

Device Driver:

- software that initializes and operates the hardware

Operating System Emulation Layer

- Operating System specific functions are kept together in this layer to make it easier to port lwIP to different OS's.

- Required functionality in the operating system or kernel:
  - One-shot timer with at least 200 ms resolution
  - Semaphore process synchronization mechanism
  - Mailbox system with:
    (1) nonblocking "post to message" queue function
    (2) blocking "receive from message" queue function

- Lightweight IP has its own buffer management system so that it can exploit potential efficiencies and also be more independent of the other parts of the system.

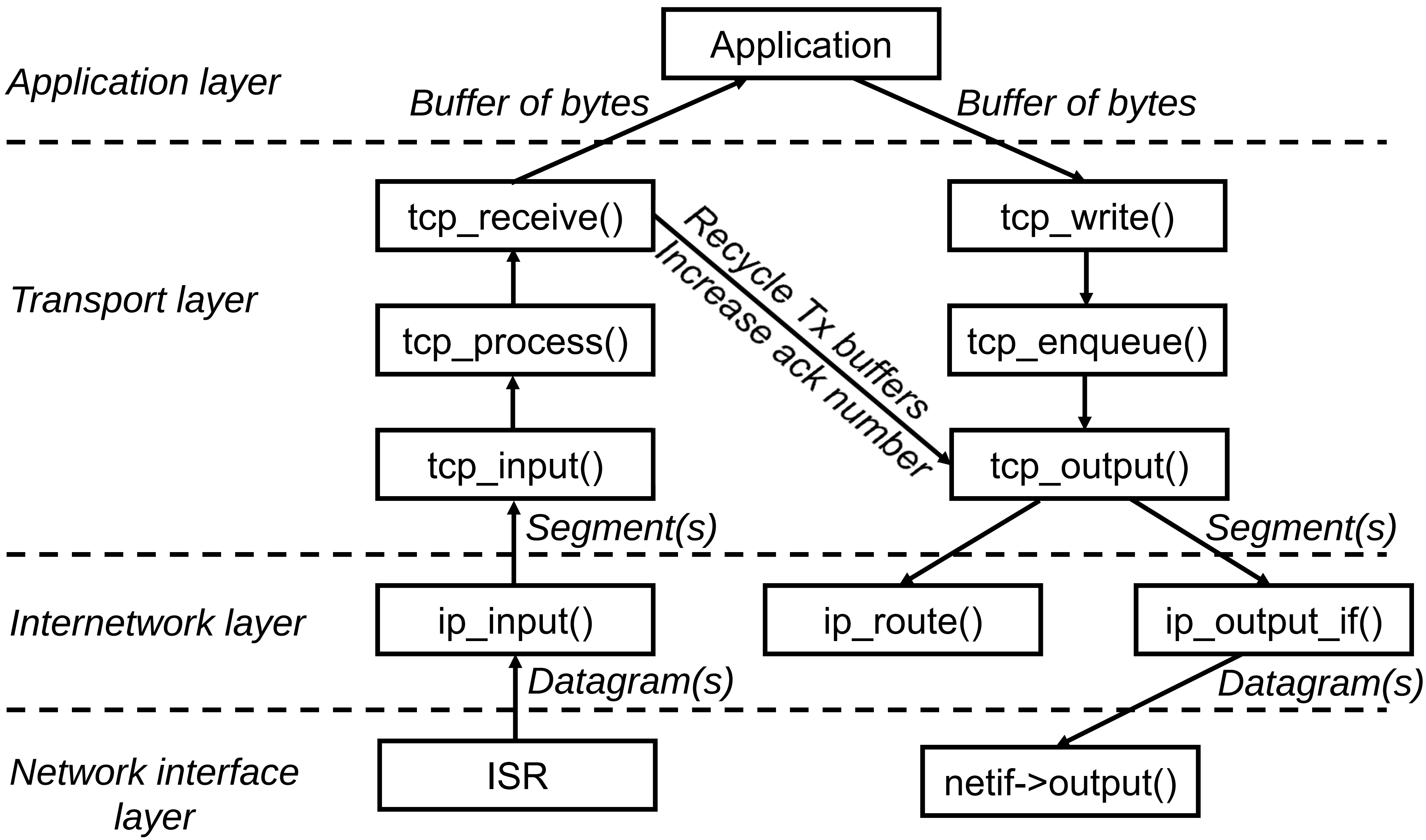
Device Driver

- A ***device driver*** is a body of software that takes care of initializing and operating a hardware device or interface. Typical components in a device driver:
  - initialization routine
  - data output routine
  - data input routine
  - interrupt service routine(s)
  - data buffers, state variables
  - query interface status

- In lwIP, as in Unix systems, there can be several device drivers available for operating different network interfaces.

- All device drivers can be used in the same way by higher-level lwIP software, so lwIP is shielded from device-specific details.

TCP/IP Data Processing in lwIP



Processing of Incoming Datagrams in lwIP

ip_input()
- verifies the IP header of the datagram
- datagram discarded if error detected
- verifies if datagram destination is host node
- passes resulting segment up to the appropriate higher layer protocol (TCP, UDP, ICMP)

tcp_input()
- verifies TCP header checksum in segments
- parses options in the TCP header
- from port number, determines TCP connection

tcp_process()
- make any state transitions in TCP protocol
- call tcp_receive() if connection ready for data

tcp_receive()
- strip off TCP header and pass data to application
- recycle any ACKed outgoing segments in buffer
- call tcp_output() if ACKed transmitted bytes allow more outgoing segments to be transmitted

Processing of Outgoing Data Bytes in lwIP

tcp_write()	- called to pass data byte buffer to TCP layer
tcp_enqueue()	- if necessary, break up data buffer into pieces - queue up the resulting TCP segments
tcp_output()	- check to see if more segments can be sent - if yes, call ip_route then ip_output_if()
ip_route()	- find the next node in the datagram's route and select the outgoing port from the present node
ip_output_if()	- assemble datagram, and call device driver
ifnet->output()	- the device driver that loads the datagram into the transmitter hardware