APPLICATION NOTE

# AT04055: Using the lwIP Network Stack

**Atmel SAM4E**

## Introduction

This application note aims at describing and understanding the lwIP stack, in order to quickly design efficient connected applications. The various lwIP APIs will be discussed in depth as well as porting the stack to any hardware like the SAM4E GMAC. Finally, detailed examples will be outlined along with the memory footprint information.

It is expected from the user to understand the basic concept described in the Open Systems Interconnection (OSI) model along with the TCP/IP protocol.

## Features

- Atmel® AT91SAM4E Ethernet Gigabit MAC (GMAC) module
  - Compatible with IEEE® 802.3 Standard
  - 10/100Mbps operation
  - MII Interface to the physical layer
  - Direct Memory Access (DMA) interface to external memory
- lwIP TCP/IP APIs
- lwIP memory management
- lwIP GMAC network interface driver
- lwIP demo for IAR™ 6.5
  - Raw HTTP basic example
  - Netconn HTTP stats example
- lwIP debugging
- lwIP optimizing SRAM footprint

## Table of Contents

# 1. LwIP Stack Overview

## 1.1 Presentation

The lightweight Internet Protocol (lwIP) is a small independent implementation of the network protocol suite that has been initially developed by Adam Dunkels.

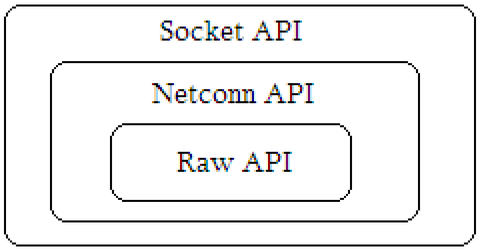
The focus of the lwIP network stack implementation is to reduce memory resource usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

lwIP supports the following protocols:

- ARP (Address Resolution Protocol)
- IP (Internet Protocol) v4 and v6
- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)
- DNS (Domain Name Server)
- SNMP (Simple Network Management Protocol)
- DHCP (Dynamic Host Configuration Protocol)
- ICMP (Internet Control Message Protocol) for network maintenance and debugging
- IGMP (Internet Group Management Protocol) for multicast traffic management
- PPP (Point to Point Protocol)
- PPPoE (Point to Point Protocol over Ethernet)

lwIP offers three different APIs designed for different purposes:

- **Raw API** is the core API of lwIP. This API aims at providing the best performances while using a minimal code size. One drawback of this API is that it handles asynchronous events using callbacks which complexify the application design.
- **Netconn API** is a sequential API built on top of the Raw API. It allows multi-threaded operation and therefore requires an operating system. It is easier to use than the Raw API at the expense of lower performances and increased memory footprint.
- **BSD Socket API** is a Berkeley like Socket implementation (Posix/BSD) built on top of the Netconn API. Its interest is portability. It shares the same drawback than the Netconn API.



lwIP is licensed under a BSD-style license: http://lwip.wikia.com/wiki/License.

lwIP source can be fetched from here: http://savannah.nongnu.org/projects/lwip.

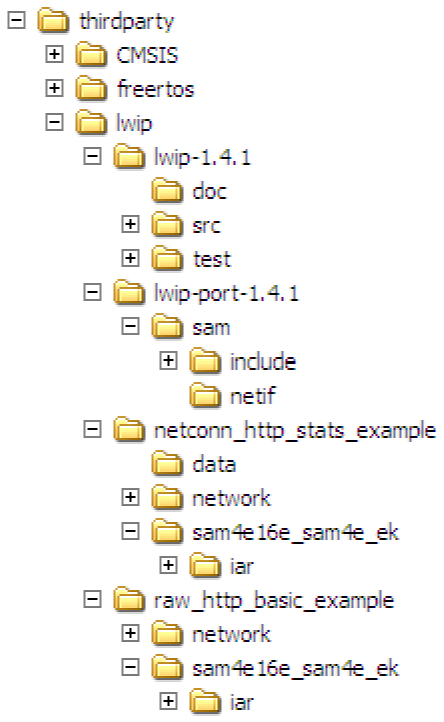
Note: lwIP is also present as a thirdparty in the Atmel Software Framework (ASF).

## 1.2 Folder Structure

Atmel provides different versions of the lwIP network stack under the thirdparty folder in the ASF. Note that in the ASF, each lwIP version comes with a port responsible for enabling hardware support for each SAM device.

lwIP example folder structure is detailed in Figure 1-1.

**Figure 1-1. LwIP Folder Structure**



- **raw_http_basic_example**: basic HTTP server example using Raw API.
  - **sam4e16e_sam4e_ek/iar**: the IAR project folder for the raw HTTP basic example.
- **netconn_http_stats_example**: advanced HTTP server example using Netconn API.
  - **sam4e16e_sam4e_ek/iar**: the IAR project folder for the Netconn HTTP stats example.
- **lwip-1.4.1/src**: lwIP source files.
  - **api**: lwIP Netconn and BSD API implementation.
  - **core**: lwIP core Raw API implementation.
- **lwip-port-1.4.1/sam**: lwIP MAC driver for lwIP. Support both standalone and RTOS modes.
  - **arch**: compiler and RTOS abstraction layers.
  - **netif**: network interface driver for SAM4E GMAC interfacing with the lwIP network stack.

A lwIP project always contains a lwIP configuration file named lwipopts.h. This file is located at the project's root directory. Note that this file does not include all the possible lwIP options. Any missing option from this configuration file can be imported (copy/paste) from the thirdparty/lwip/lwip-1.4.1/src/include/lwip/opt.h file.

# 2. lwIP TCP API

## 2.1 Raw API

The Raw API is a non-blocking, event-driven API designed to be used without an operating system that implements zero-copy send and receive. This API is also used by the core stack for interaction between the various protocols. It is the only API available when running lwIP without an operating system.

Applications using the Raw API implement callback functions, which are invoked by the lwIP core when the corresponding event occurs. For instance, an application may register to be notified via a callback function for events such as incoming data available (tcp_recv), outgoing data sent (tcp_sent), error notifications (tcp_err), etc. An application should provide callback functions to perform processing for any of these events.

Table 2-1 provides a summary of the Raw API functions for TCP.

**Table 2-1. TCP Raw API Functions**

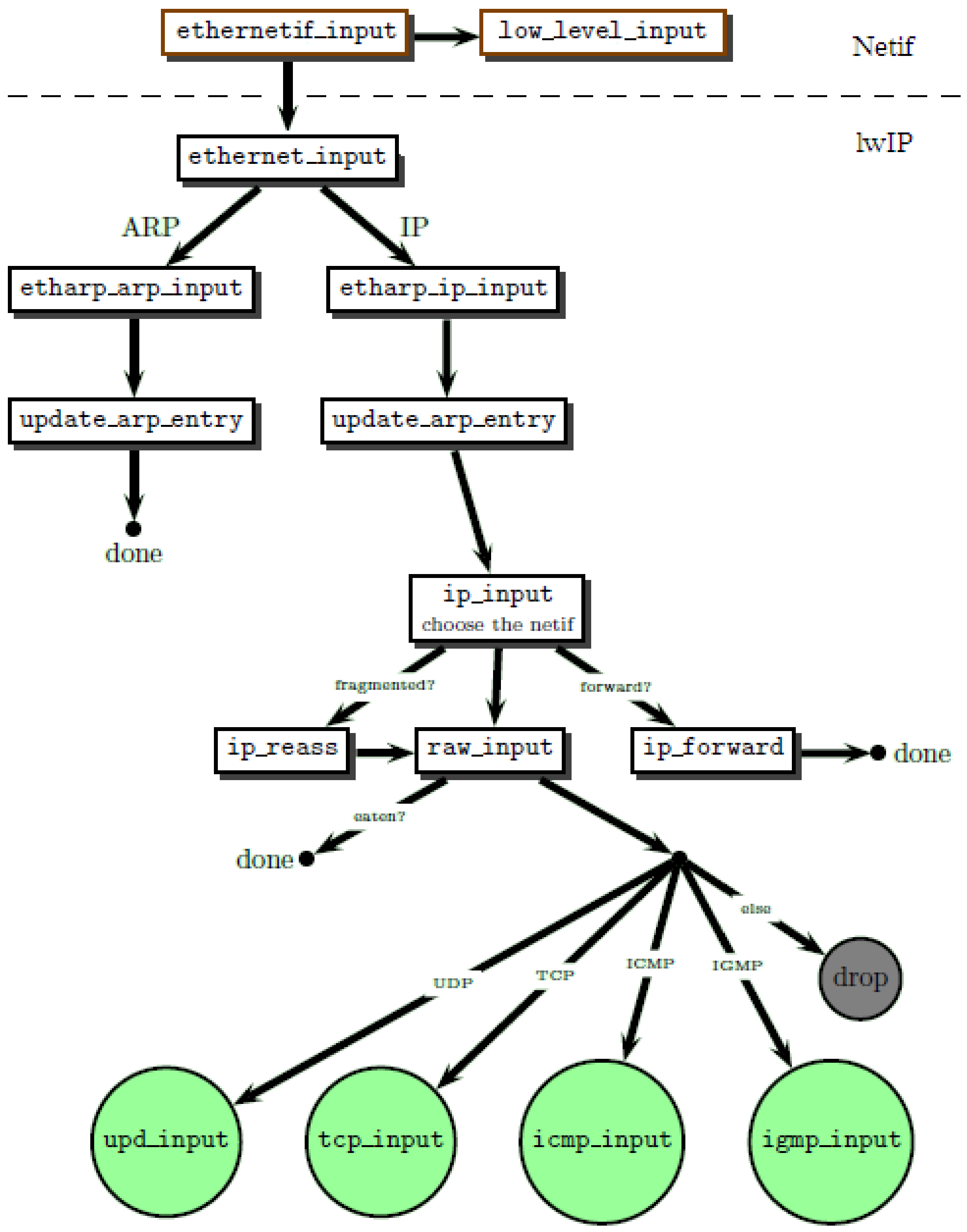
| | API function | Description |
|---|---|---|
| TCP connection setup | tcp_new | Creates a new connection PCB (Protocol Control Block). A PCB is a structure used to store connection status. |
| | tcp_bind | Binds the pcb to a local IP address and port number. |
| | tcp_listen | Commands a pcb to start listening for incoming connections. |
| | tcp_accept | Sets the callback function to call when a new connection arrives on a listening connection. |
| | tcp_accepted | Inform lwIP that an incoming connection has been accepted. |
| | tcp_connect | Connects to a remote TCP host. |
| Sending TCP data | tcp_write | Queues up data to be sent. |
| | tcp_sent | Sets the callback function that should be called when data has successfully been sent and acknowledged by the remote host. |
| Receiving TCP data | tcp_recv | Sets the callback function that will be called when new data arrives. |
| | tcp_recved | Informs lwIP core that the application has processed the data. |
| Application polling | tcp_poll | Specifies the polling interval and the callback function that should be called to poll the application. This feature can be used to check if there are some pending data to be sent or if the connection needs to be closed. |
| Closing and aborting connections | tcp_close | Closes a TCP connection with a remote host. |
| | tcp_err | Sets the callback function to call when a connection is aborted because of an error. |
| | tcp_abort | Aborts a TCP connection. |

To enable the lwIP stack, the user application has to perform two functions calls from the main program loop:

- ethernetif_input() to treat incoming packets (function defined in the network interface GMAC driver)
- timers_update() to refresh and trigger the lwIP timers (defined in the user application, typically under the network folder)

Figure 2-1 shows the lwIP receive flow from the ethernetif_input() function (lwip-port-1.4.1/src/sam/netif/sam4e_gmac.c) to the appropriate input protocol function in the lwIP core (lwip-1.4.1/src/core/ files). The ethernetif_input() function should typically be called from the main program loop.

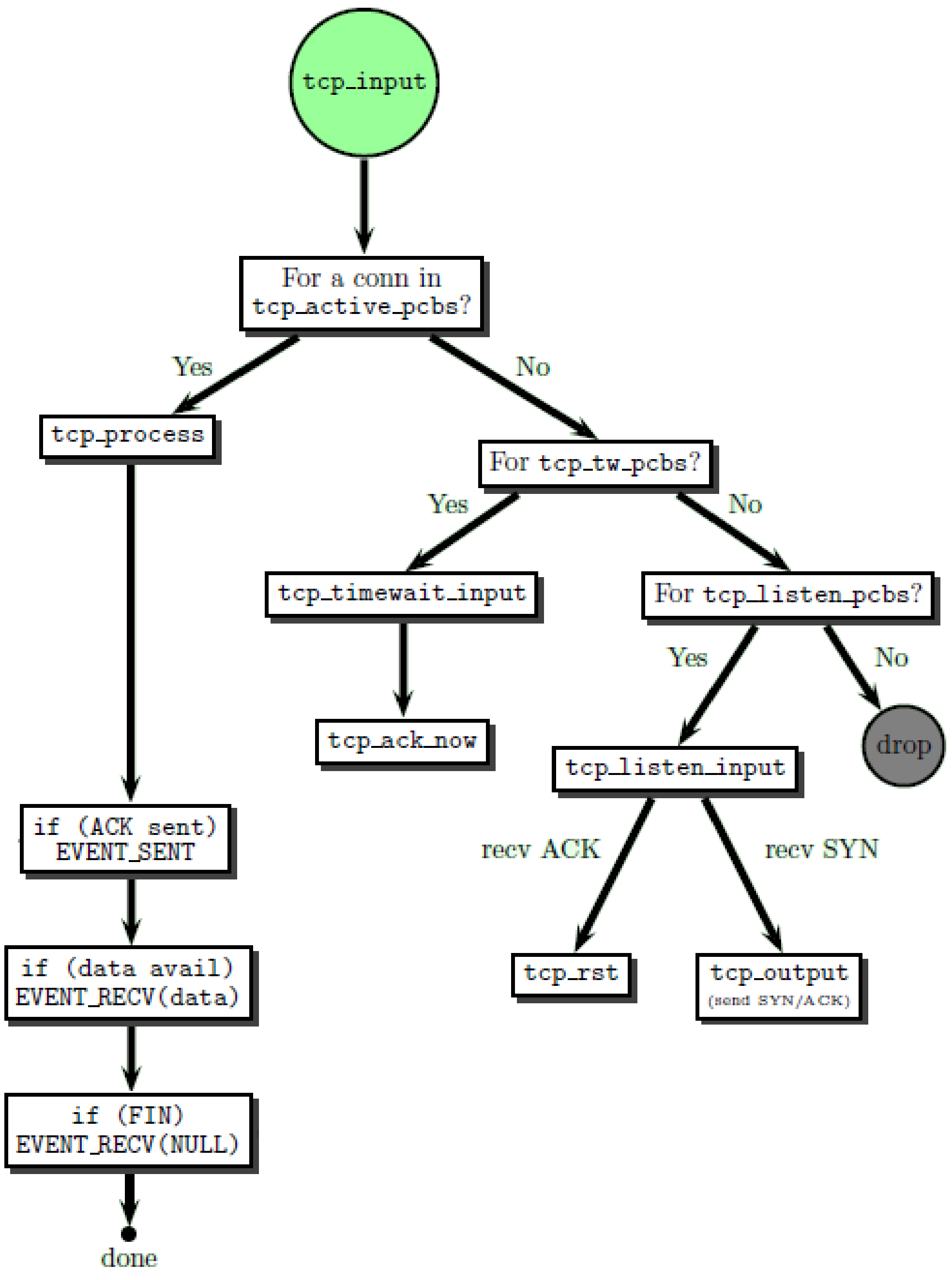
Figure 2-1. lwIP Receive Flow



The ip_input() function checks for a valid IP checksum and ensures that the packet is addressed to the device.

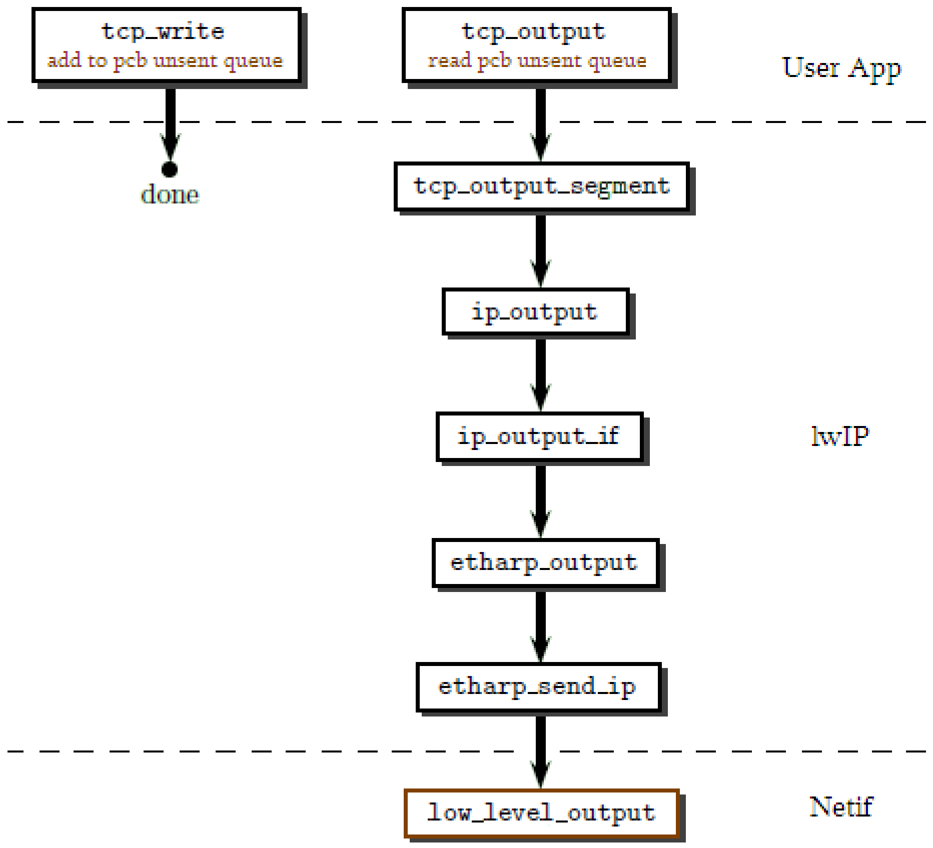
The raw_input() function tries to find a raw protocol control block (PCB) to handle the incoming packet. Raw PCBs are used to implement custom network protocols. If there is no raw PCB to be used, the appropriate input protocol function is called using the protocol field from the IP header.

**Figure 2-2. lwIP TCP Input Flow**



As shown in Figure 2-2, the tcp_input() function tries to find the PCB keeping track of the connection used by the incoming packet (using IP and port numbers). The TCP checksum is verified, then depending on the incoming packet, the tcp_input() function will eventually inform the user application on specific events (like data sent, data received, etc) using the previously registered callbacks. See Table 2-1 for some functions to register these callbacks.

**Figure 2-3. lwIP TCP Output Flow**



The lwIP network stack provides the tcp_write() function for sending data to a remote host, as shown in Figure 2-3. It takes a pointer to the PCB structure (representing the active connection to write to), data buffer, data size and API flags as parameters. This function attempts to build TCP segments from the user data buffer. A TCP segment is referred to as a TCP header and a user data. This segment is made of several PBUFs which allow the user data buffer to be either referenced or copied depending on the *TCP_WRITE_FLAG_COPY* flag. If the user data size is superior to the defined TCP_MSS, the data buffer spreads onto several TCP segments. The TCP segments are then added to the PCB's unsent queue. Alternatively, the user data can be directly prepended to the last enqueued TCP segment if its remaining size does not exceed the *TCP_MSS* value. Note that at this stage no data has been sent over the Ethernet link.

The TCP segments are only sent when a call to the tcp_output() function is made, as shown in Figure 2-3. This function is also automatically triggered by lwIP in the following cases:

- Inside the tcp_input() function (when TCP acknowledgement has to be sent right away)
- Inside the slow and fast timers (where retransmitting TCP segments can be required)

At this stage, the TCP segment gets encapsulated with the IP header (ip_output() function) and Ethernet header (etharp_output() function). Finally, the Ethernet frame is sent to the GMAC IP via the low_level_output() function located in the lwIP netif driver (lwip-port-1.4.1/src/sam/netif/sam4e_gmac.c).

When using the Raw API in a multithreaded environment, beware that there is no protection against concurrent access. Consequently, all the network part of the application must remain in a single thread context.

## 2.2 Netconn API

The Netconn API is a sequential API designed to be used with an operating system while preserving the zero-copy functionality. This API is built on top of the Raw API and makes the stack easier to use compared to the event-driven Raw API.

A Netconn API based program would typically use the following threads:

- tcpip-thread: the lwIP core thread which actually makes use of the Raw API
- GMAC: the netif driver thread in charge of passing Ethernet frame from the GMAC IP to the tcpip-thread
- One or more user application threads performing open/read/write/close operations on Netconn connections

The above threads are communicating using message passing which is fully handled by the Netconn API.

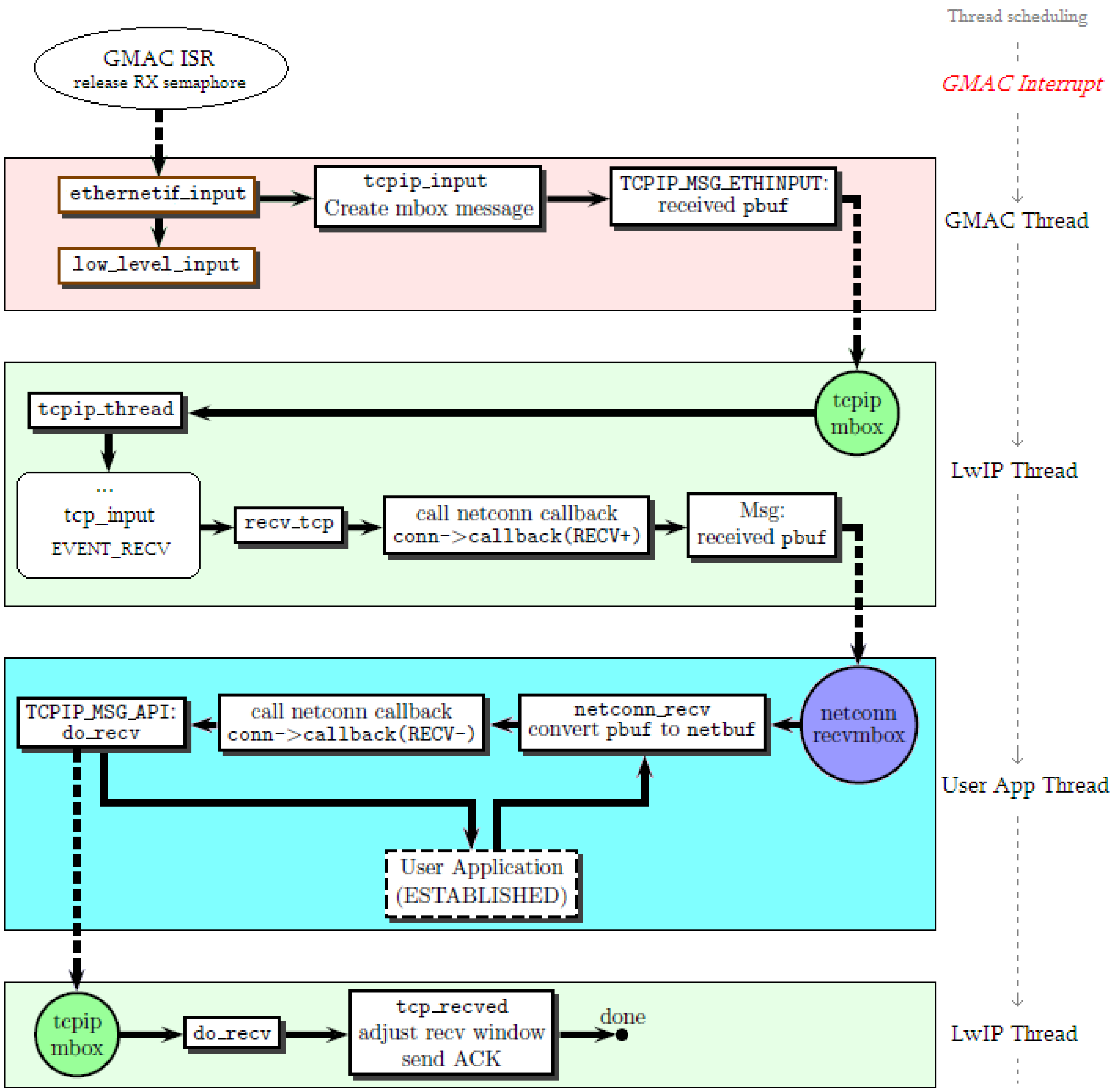
Table 2-2 provides a summary of the Netconn API functions for TCP.

**Table 2-2. TCP Netconn API Functions**

| API function | Description |
|---|---|
| netconn_new | Creates a new Netconn connection. |
| netconn_bind | Binds a Netconn structure to a local IP address and port number. |
| netconn_listen | Sets a TCP Netconn connection into listen mode. |
| netconn_accept | Accepts an incoming connection on a listening TCP Netconn connection. |
| netconn_connect | Connects to a remote TCP host using IP address and port number. |
| netconn_recv | Receives data from a Netconn connection. |
| netconn_write | Sends data on a connected TCP Netconn connection. |
| netconn_close | Closes a TCP Netconn connection without deleting it. |
| netconn_delete | Deletes an existing Netconn connection. |

Figure 2-4 gives an overview of an input TCP packet processing while using the Netconn API. Depending on the thread priorities, a minimum of 4 context switches is required to process one single TCP packet. If the user application requires maximum performances Raw API should be considered instead of the Netconn API.

**Figure 2-4. lwIP TCP Input Flow using the Netconn API (RTOS based)**



As opposed to the Raw API approach, the GMAC driver does not process the TCP packet directly. Instead by calling the tcpip_input() function it notifies the lwIP core thread using the tcpip "mbox" mailbox that a packet is ready for processing. Then the lwIP core thread wakes up, reads the tpcip "mbox" message and starts the packet processing using the Raw API (calling ethernet_input() function, as shown in Figure 2-1). When a valid TCP packet is found, the lwIP core thread notifies the corresponding Netconn socket using the "recvmbox" mailbox.

From the user application point of view, when calling the netconn_recv() function, the user application thread waits for a message in the recvmbox to know if a TCP input packet has arrived. The user application can wait forever or for the specified amount of time if *LWIP_SO_RCVTIMEO* has been defined in the configuration file. When the "recvmbox" message is available, the user application thread wakes up and sends a notification message to the tcpip "mbox" to give a chance to acknowledge the packet and to adjust the receive window if necessary. During that time the user application thread waits for a notification semaphore only released by the lwIP core thread when the operation is completed. Finally, the netconn_recv() function returns the netbuf structure containing the TCP packet data to the user application.

The following mailbox sizes must be defined in the lwIP configuration file when using the Netconn API:

- **TCPIP_MBOX_SIZE**: size of the core tcpip mailbox
- **DEFAULT_ACCEPTMBOX_**: SIZE size of the accept mailbox
- **DEFAULT_TCP_RECVMBOX_**: SIZE size of the TCP recv mailbox

## 2.3 BSD Socket API

The lwIP socket API is built on top of the Netconn API and offers portability for BSD socket based applications. Table 2-3 provides a summary of the Socket API functions.

**Table 2-3. Socket API Functions**

| API function | Description |
|---|---|
| socket | Creates a new socket. |
| bind | Binds a socket to a local IP address and port number. |
| listen | Listens for incoming connections on socket. |
| accept | Accepts an incoming connection on a listening socket. |
| connect | Connects a socket to a remote host using IP address and port number. |
| read | Reads data from a socket. |
| write | Writes data to a socket. |
| close | Closes a socket. |

To enable BSD-style socket support; the lwIP configuration file must define *LWIP_SOCKET* and *LWIP_COMPAT_SOCKETS*. For more information about these defines please refer to src/include/lwip/opt.h configuration file.

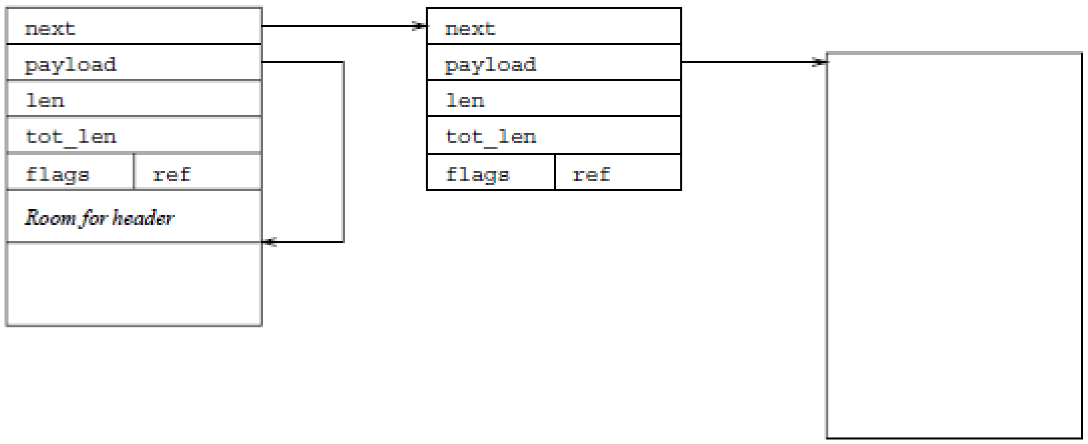
# 3. lwIP Memory Management

## 3.1 Pbuf Structure

The lwIP pbuf data structure is a linked list of buffers designed to represent one network packet. Indeed, when the user application performs a tcp_write() function call with a user data, the lwIP stack prepends (and possibly appends other pbufs) to encapsulate the various protocol headers to build the corresponding packet. Depending on the pbuf allocation the lwIP stack can leave enough room to add the protocol header, or it can simply add another pbuf in the pbuf's linked list. Using the pbuf data structure, headers can be added without copying the entire buffer.

This data structure provides support for allocating dynamic memory to hold packet data, and for referencing data in static memory. A pbuf linked list is refered to as a pbuf chain, hence a complete packet may span over several pbufs.

**Figure 3-1. A PBUF RAM Chained with a PBUF ROM that has Data in External Memory**



The pbuf structure has the following fields:

- **next**: pointer to the next pbuf element in the pbuf linked list
- **payload**: pointer to the data
- **len**: length of the payload
- **tot_len**: sum of the len fields of the pbuf chain
- **flags**: pbuf type
- **ref**: reference count. A pbuf can only be freed if its reference count is zero

The pbuf type can be one of the following:

- **PBUF_POOL**: a number of *PBUF_POOL_SIZE* pbufs are statically pre allocated with a fixed size of *PBUF_POOL_BUFSIZE* (defined in the lwIP configuration file). This is the pbuf type used for packet reception as it provides the fastest allocation.
- **PBUF_RAM**: pbuf are dynamically allocated from a contiguous memory area. The *PBUF_RAM* allocation is slower than the *PBUF_POOL* and can lead to memory fragmentation.
- **PBUF_ROM**: pbuf used to pass constant data by reference.
- **PBUF_REF**: pbuf used to pass data from the user application by reference. As the pointed data is not constant, it may be internally copied into *PBUF_RAM* if lwIP needs to enqueue this pbuf.

## 3.2 Pbuf API

Table 3-1 shows a brief summary of the pbuf API. Implementation can be found in the src/core/pbuf.c file.

**Table 3-1. Pbuf API Functions**

| API function | Description |
|---|---|
| pbuf_alloc | Allocates a new pbuf. |
| pbuf_ref | Increments the reference count of a pbuf. |
| pbuf_free | Decrements the pbuf reference count. When reference equals zero, the pbuf is de-allocated. |
| pbuf_clen | Returns the number of pbuf in a pbuf chain. |
| pbuf_cat | Concatenates two pbufs. |
| pbuf_take | Copies the provided data into a pbuf. |

When using the Netconn API, pbufs are wrapped in a netbuf structure. The netbuf API implementation can be found in the thirdparty/lwip/lwip-1.4.1/src/api/netbuf.c file.

# 4. Developing a Network Interface for lwIP

## 4.1 Abstraction Layers

The lwIP network stack has defined a set of function prototypes to interface with MAC hardware. The network interface drivers are located in the thirdparty/lwip/lwip-port-1.4.1/sam/netif folder. See Table 4-1 for the function list and summary.

**Table 4-1. Network Interface Functions for the lwIP Network Stack**

| API function | Description |
|---|---|
| ethernetif_init | Calls low_level_init() function and initializes the netif structure. |
| ethernetif_input | Calls low_level_input() function to read a packet from the MAC hardware and passes it to the lwIP input function. |
| low_level_init | Initializes the MAC hardware. |
| low_level_input | Reads one packet from the MAC hardware. |
| low_level_output | Writes one packet to the MAC hardware. |

lwIP has been designed to be compiled on many different platforms; hence the thirdparty/lwip/lwip-port-1.4.1/sam/include/arch/cc.h file provides a compiler abstraction level.

lwIP also implements one other abstraction level for using RTOS services when using the Netconn or Socket APIs. This abstraction layer is implemented in the thirdparty/lwip/lwip-port-1.4.1/sam/sys_arch.c file. Current implementation only supports FreeRTOS™.

## 4.2 GMAC Network Interface

The GMAC driver associates a set of functions to drive the GMAC hardware and to manage the Ethernet physical layer (PHY).

The source files are located as following:

- sam/drivers/gmac: contains functions to drive the GMAC peripheral and includes generic PHY routines
- sam/components/ethernet_phy/ksz8051mnl: adds support to the Micrel PHY used on the SAM4E-EK
- thirdparty/lwip/lwip-port-1.4.1/sam/netif: implements lwIP interface and GMAC driver logic

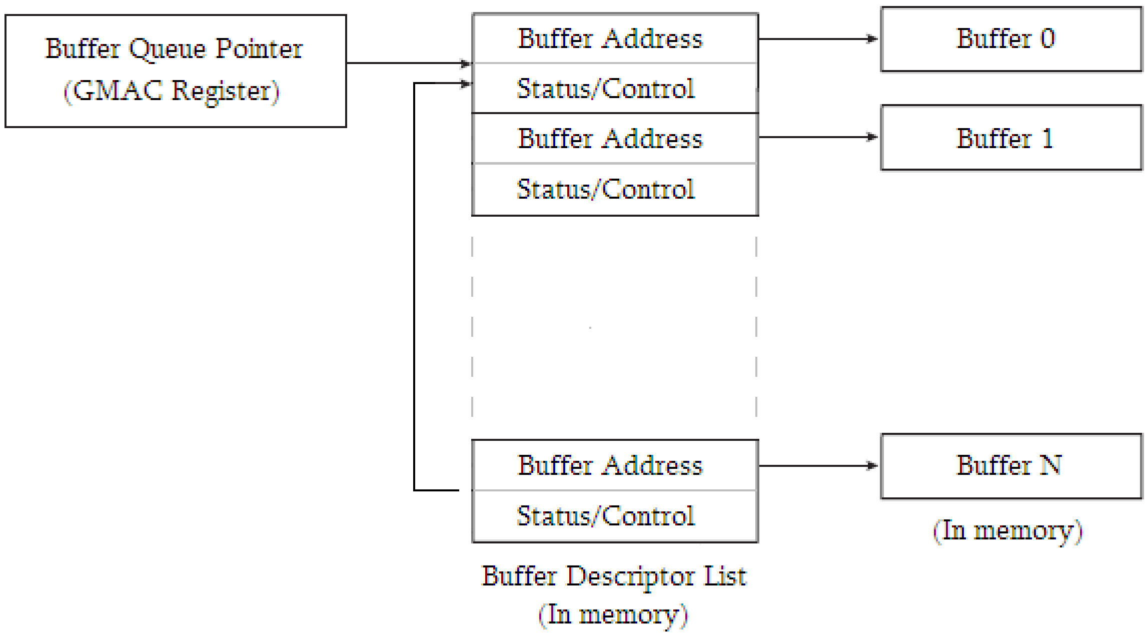
Note: The conf_eth.h file is used to configure the GMAC driver (number of RX/TX buffers, IP settings and PHY mode).

### 4.2.1 DMA Programming

The GMAC IP can send or receive Ethernet frames by performing Direct Memory Access from memory (typically SRAM) to internal FIFO. Setting up the DMA transfer requires using two sets of buffer descriptors and configuring the Buffer Queue Pointer; one for data receive and one for data transmit. One buffer descriptor being a pointer to the actual send/receive buffer and a 32-bit word for status or control.

Figure 4-1 illustrates the GMAC DMA configuration.

Figure 4-1. GMAC DMA Descriptor List and Buffers



Note that the SAM4E AHB bus matrix does not allow transfer from flash to GMAC DMA.

The GMAC receive /transmit descriptor lists and buffers are implemented in the gmac_device structure (thirdparty/lwip/lwip-port-1.4.1/sam/netif/sam4e_gmac.c) as following:

```
/** Pointer to Rx descriptor list (must be 8-byte aligned) */
gmac_rx_descriptor_t rx_desc[GMAC_RX_BUFFERS];
/** Pointer to Tx descriptor list (must be 8-byte aligned) */
gmac_tx_descriptor_t tx_desc[GMAC_TX_BUFFERS];
/** RX pbuf pointer list */
struct pbuf *rx_pbuf[GMAC_RX_BUFFERS];
/** TX pbuf pointer list */
struct pbuf *tx_pbuf[GMAC_TX_BUFFERS];
```

The number of receive /transmit descriptors and buffers can be changed in file conf_eth.h by setting the *GMAC_RX_BUFFERS/ GMAC_TX_BUFFERS* values.

RX DMA buffers are pointers to pbuf structures allowing zero-copy transfers between the GMAC network interface and the lwIP network stack.

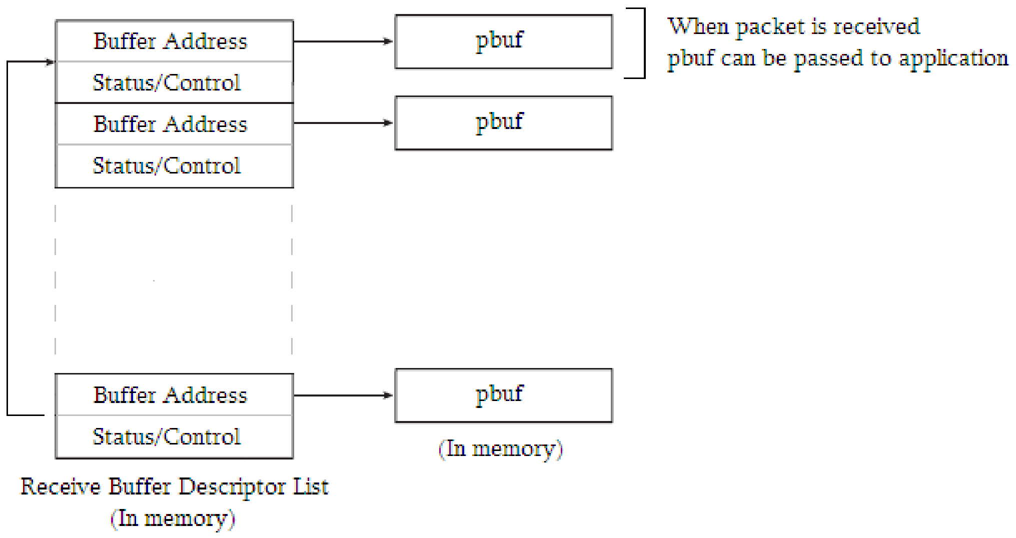
### 4.2.2 Receive Buffers

Zero-copy RX buffers can improve system performances over copied buffers when transferring large amounts of data. It also uses less memory as it can directly use pbufs allocated by the lwIP network stack; hence simplifying the memory configuration.

To enable packet reception, pbufs are pre-allocated during the GMAC network interface initialization stage. The GMAC DMA receive buffer size is fixed to 1536 bytes; meaning that one receive buffer will handle exactly one TCP packet.

When a packet is received, the GMAC IP will place the received data into the pbuf's payload area. Then the network interface driver removes this pbuf from its descriptor and passes it to the lwIP network stack without an extra copy. The network interface does not keep track this pbuf as the lwIP network stack will free this resource once the data has been consumed by the user application. Finally, a new pbuf is allocated for the previous buffer descriptor which is now updated and ready to use.

**Figure 4-2. Zero-copy Receive Descriptor List and Buffers**



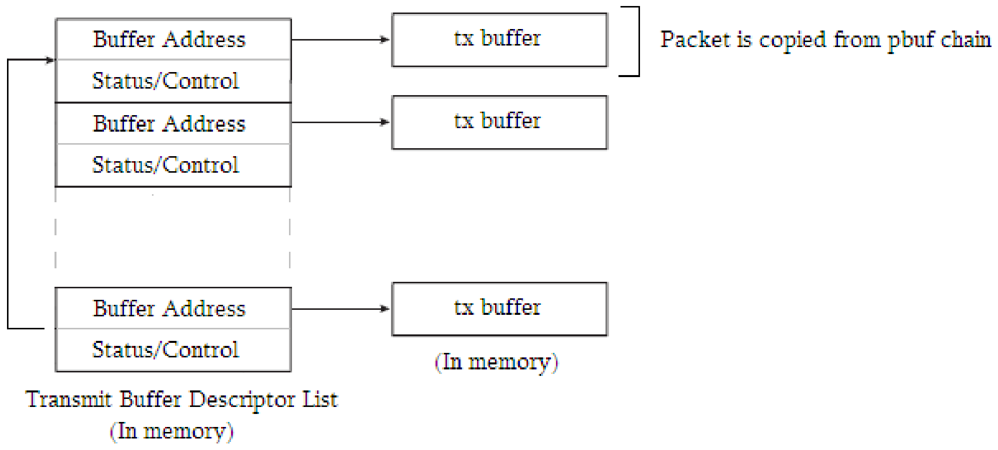
Beware that the amount of memory used by receive (pbuf) buffers is constant and equals to *GMAC_RX_BUFFERS * PBUF_POOL_BUFSIZE*. The lwIP total memory size *MEM_SIZE* must be set accordingly.

## 4.2.3 Transmit Buffers

TX buffers are statically pre allocated with the maximum packet size in the GMAC network interface driver. The pbuf chain passed from the network layer is fully copied into a TX buffer before sending.

Transmit descriptors should remain free most of the time. Hence, *GMAC_TX_BUFFERS* can be defined low to reduce lwIP memory requirements.

**Figure 4-3. Transmit Descriptor List and Buffers**



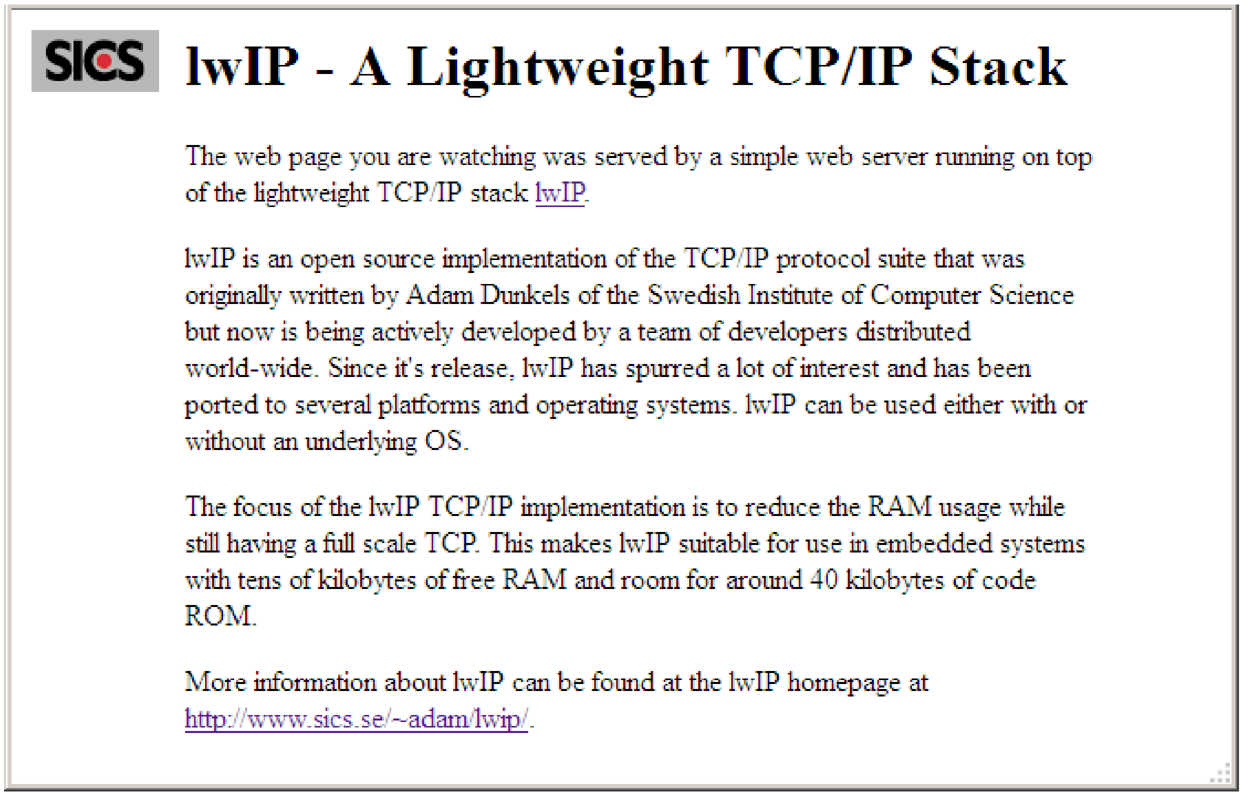
# 5. lwIP Demo Applications

## 5.1 Basic HTTP Server using the Raw API

This demo is a basic HTTP server implementation that can serve one request at a time. The project file for IAR is located in the thirdparty/lwip/lwip-1.4.1/raw_http_basic_example/sam4e16e_sam4e_ek/iar folder.

The HTTP server home page can be accessed using any browser at http://192.168.0.100.

**Figure 5-1. Home Page of the Raw HTTP Basic Example**



lwIP - A Lightweight TCP/IP Stack

The web page you are watching was served by a simple web server running on top of the lightweight TCP/IP stack lwIP.

lwIP is an open source implementation of the TCP/IP protocol suite that was originally written by Adam Dunkels of the Swedish Institute of Computer Science but now is being actively developed by a team of developers distributed world-wide. Since it's release, lwIP has spurred a lot of interest and has been ported to several platforms and operating systems. lwIP can be used either with or without an underlying OS.

The focus of the lwIP TCP/IP implementation is to reduce the RAM usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

More information about lwIP can be found at the lwIP homepage at http://www.sics.se/~adam/lwip/.

### 5.1.1 Code Overview

The main function (located in the thirdparty/lwip/lwip-1.4.1/raw_http_basic_example/ raw_http_basic_example.c file) performs the following initialization steps:

- Configure system clock and pins
- Configure the UART console
- Initialize the lwIP network stack
- Initialize the HTTP server socket
- Update the Ethernet task in a while loop

```
int main(void)
{
          /* Initialize the SAM system. */
          sysclk_init();
          board_init();

          /* Configure debug UART */
          configure_console();

          /* Print example information. */
          puts(STRING_HEADER);

          /* Bring up the ethernet interface & initialize timer0, channel0. */
          init_ethernet();

          /* Bring up the web server. */
          httpd_init();

          /* Program main loop. */
          while (1) {
              /* Check for input packet and process it. */
              ethernet_task();
          }
}
```

The lwIP network stack initialization is done in the init_ethernet function (located in the thirdparty/lwip/lwip-1.4.1/raw_http_basic_example/ network/ethernet.c file). It performs the following steps:

- Initialize the lwIP network stack to operate in Raw API mode (non RTOS)
- Configure the network interface driver
- Configure a 1ms timer counter (TC) to handle the lwIP timers

```
void init_ethernet(void)
{
          /* Initialize lwIP. */
          lwip_init();

          /* Set hw and IP parameters, initialize MAC too. */
          ethernet_configure_interface();

          /* Initialize timer. */
          sys_init_timing();
}
```

The ethernet_configure_interface() function (located in the thirdparty/lwip/lwip-1.4.1/raw_http_basic_example/ network/ethernet.c file) is detailed below:

```
static void ethernet_configure_interface(void)
{
       …
       /* Add data to netif */
       if (NULL == netif_add(&gs_net_if, &x_ip_addr, &x_net_mask, &x_gateway, NULL,
              ethernetif_init, ethernet_input)) {
          LWIP_ASSERT("NULL == netif_add", 0);
       }
       /* Make it the default interface */
       netif_set_default(&gs_net_if);

       /* Setup callback function for netif status change */
       netif_set_status_callback(&gs_net_if, status_callback);

          /* Bring it up */
#if defined(DHCP_USED)
          /* DHCP mode. */
          if (ERR_OK != dhcp_start(&gs_net_if)) {
             LWIP_ASSERT("ERR_OK != dhcp_start", 0);
          }
          printf("DHCP Started\n");
#else
          /* Static mode. */
          netif_set_up(&gs_net_if);
          printf("Static IP Address Assigned\n");
#endif
}
```

If the DHCP_USED macro is defined in the lwIP configuration file, a DHCP client is started to fetch an IP address, else the static IP address defined in the thirdparty/lwip/raw_http_basic_example /sam4e16e_sam4e_ek/conf_eth.h file is used.

A pointer to a status_callback function is used to print the device IP address on the UART console once lwIP configuration is done.

The actual HTTP server initialization is made from the main() function by calling the httpd_init() function. This function instanciates a new TCP PCB and listen for incoming connection of the HTTP port 80. The tcp_accept() function is used to define a callback to the http_accept() function once an incoming connection is detected on port 80.

```
void httpd_init(void)
{
          struct tcp_pcb *pcb;

          pcb = tcp_new();
          tcp_bind(pcb, IP_ADDR_ANY, 80);
          pcb = tcp_listen(pcb);
          tcp_accept(pcb, http_accept);
}
```

The following http_accept() function is called from the lwIP network state to initialize the state of the connection. It is mainly used to allocate a user data structure and register callbacks for the desired events:

- **tcp_recv**: to wait for data to become available on the socket
- **tcp_err**: to free the user data structure when the connection is lost
- **tcp_pool**: to pool the connection at the specified time interval when the connection is idle (not receiving or sending). It is here used both as a watchdog timer for killing the connection if it has stayed idle for a too long (retries condition), and as a method for waiting memory to become available. For instance, if a call to tcp_write() has failed because memory wasn't available, the tcp_pool callback will try to perform another tcp_write() call (via the http_send_data() function).

```
static err_t http_accept(void *arg, struct tcp_pcb *pcb, err_t err)
{
          struct http_state *hs;
          …

          tcp_setprio(pcb, TCP_PRIO_MIN);

          /* Allocate memory for the structure that holds the state of the
          connection. */
          hs = (struct http_state *)mem_malloc(sizeof(struct http_state));

          if (hs == NULL) {
             return ERR_MEM;
          }

          /* Initialize the structure. */
          hs->file = NULL;
          hs->left = 0;
          hs->retries = 0;

          /* Tell TCP that this is the structure we wish to be passed for our
          callbacks. */
          tcp_arg(pcb, hs);

          /* Tell TCP that we wish to be informed of incoming data by a call
          to the http_recv() function. */
          tcp_recv(pcb, http_recv);
          tcp_err(pcb, http_conn_err);
          tcp_poll(pcb, http_poll, 4);
          return ERR_OK;
}
```

Once the http_accept() callback has been performed; the socket will trigger the http_recv callback as soon as data has arrived.

The http_recv() function checks for a valid data packet, then it calls the tcp_recved() function to let the lwIP network stack to know that data has been received and that an acknowledge should be sent. Then, if the pbuf contains the GET command, the user data structure is filled with a pointer to the index.html data and the transfert begins with an http_send_data() call.

```
static err_t http_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err)
{
          …
          hs = arg;

          if (err == ERR_OK && p != NULL) {
             /* Inform TCP that we have taken the data. */
             tcp_recved(pcb, p->tot_len);

             if (hs->file == NULL) {
                data = p->payload;

                if (strncmp(data, "GET ", 4) == 0) {
                   …
                   if (*(char *)(data + 4) == '/' &&
                         *(char *)(data + 5) == 0) {
                      fs_open("/index.html", &file);
                   }
                   …
                   hs->file = file.data;
                   hs->left = file.len;
                   pbuf_free(p);
                   http_send_data(pcb, hs);

                   /* Tell TCP that we wish be to informed of data that has been
                   successfully sent by a call to the http_sent() function. */
                   tcp_sent(pcb, http_sent);
                } else {
                   …
                }
             } …
          }
          if (err == ERR_OK && p == NULL) {
             http_close_conn(pcb, hs);
          }
          return ERR_OK;
}
```

Note: The tcp_sent() function is used to register the http_sent callback once data has been successfully sent.

```
static void http_send_data(struct tcp_pcb *pcb, struct http_state *hs)
{
          err_t err;
          u32_t len;

          /* We cannot send more data than space available in the send buffer. */
          if (tcp_sndbuf(pcb) < hs->left) {
             len = tcp_sndbuf(pcb);
          } else {
             len = hs->left;
          }

          do {
             /* Use copy flag to avoid using flash as a DMA source (forbidden). */
             err = tcp_write(pcb, hs->file, len, TCP_WRITE_FLAG_COPY);
             if (err == ERR_MEM) {
                len /= 2;
             }
          } while (err == ERR_MEM && len > 1);

          if (err == ERR_OK) {
             hs->file += len;
             hs->left -= len;
          }
}
```

The http_send_data() function is used to send data via the socket connection. Since the amount of data to send can be very important, this function attempts to fill TCP send buffer. The user data structure is then updated to save the remaining amount of data to be transmitted.

When the packet is successfully transmitted, the previously registered http_sent() callback function is called, which in turns call the http_send_data() function as long as data is waiting to be transmitted.

```
static err_t http_sent(void *arg, struct tcp_pcb *pcb, u16_t len)
{
          struct http_state *hs;

          LWIP_UNUSED_ARG(len);

          hs = arg;

          hs->retries = 0;

          if (hs->left > 0) {
             http_send_data(pcb, hs);
          } else {
             http_close_conn(pcb, hs);
          }

          return ERR_OK;
}
```

When the complete data has been sent, the http_close_conn() properly closes the connection to the remote host.

### 5.1.2 Memory Footprint

The memory footprint information in Table 5-1 has been obtained using IAR 6.50.5 compiler with high optimization for size.

**Table 5-1. HTTP Raw Basic Example Memory Footprint**

| Modules | Flash (bytes) | | SRAM (bytes) |
|---|---|---|---|
| | RO code | RO data | RW data |
| GMAC+PHY driver | 1804 | 0 | 4628 |
| lwIP stack | 16154 | 46 | 11443 |
| SAM4E other drivers | 1336 | 252 | 20 |
| User application | 1580 | 3300 | 96 |
| Total | 20874 | 3598 | 16187 |
| Others (libc, stack, etc) | 5960 | 40 | 1028 |
| Grand Total | 26834 | 3638 | 17215 |

The following memory configuration was used for the lwIP network stack:

- 3 TX buffers of 1536 bytes for the GMAC driver (conf_eth.h)
- 4 buffers of 1536 bytes for the lwIP buffer pool (lwipopts.h)
- 4K for the lwIP heap memory (lwipopts.h)

Note that this memory footprint information is not optimal and can be reduced depending on your requirements.

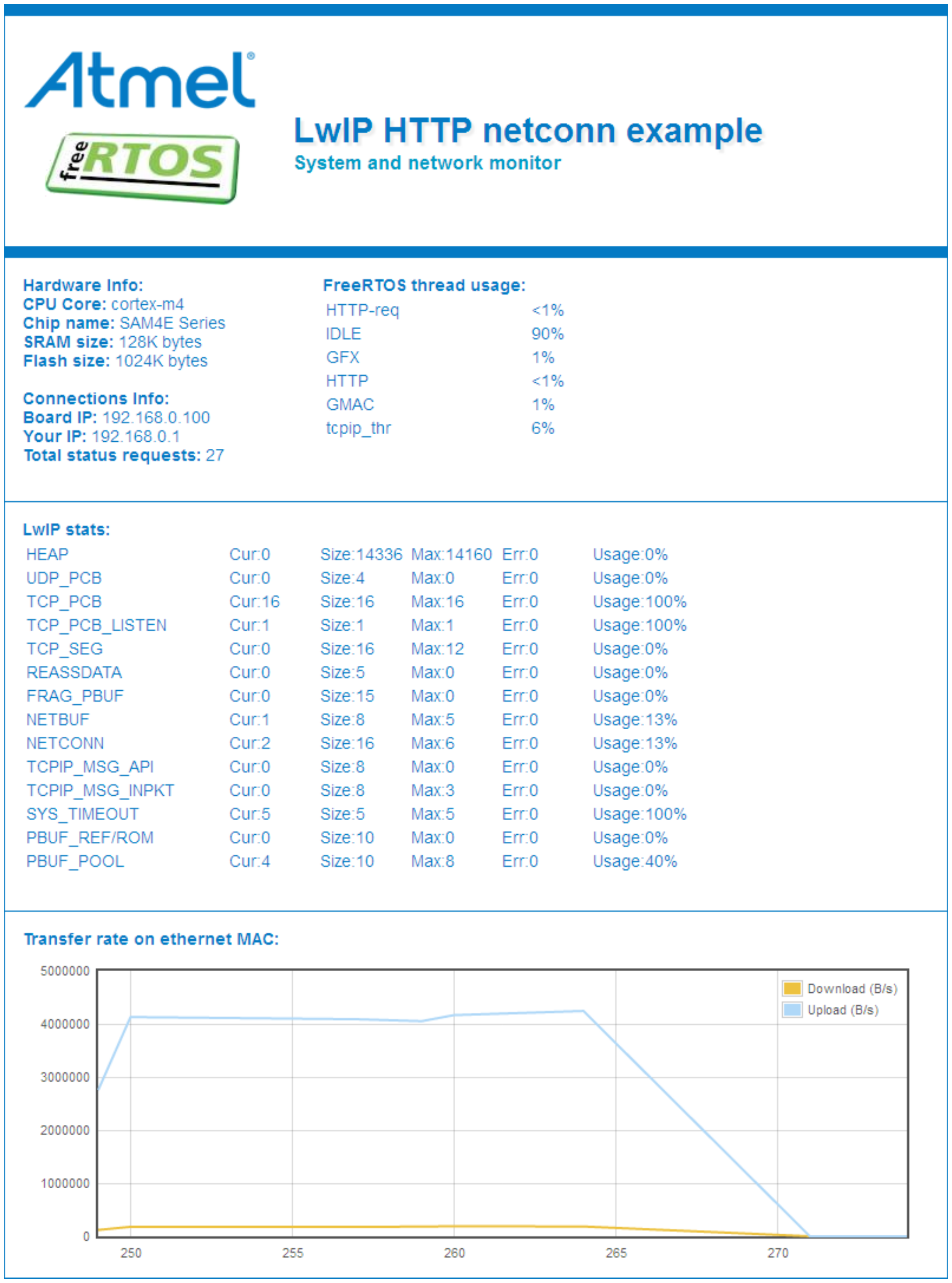
## 5.2 HTTP Stats Server using the Netconn API

This demo is FreeRTOS based and demonstrates how to develop an HTTP server that can serve several requests at the same time using the lwIP Netconn API. The server homepage provides several system informations like FreeRTOS thread usage, lwIP memory status and GMAC transfer rate in real time.

The project file for IAR is located in the thirdparty/lwip/lwip-1.4.1/netconn_http_stats_example/sam4e16e_sam4e_ek/iar folder.

The example home page can be accessed using any browser at http://192.168.0.100 (in fixed IP mode).

**Figure 5-2. Home Page of the Netconn HTTP Stats Example**



LwIP HTTP netconn example
System and network monitor

**Hardware Info:**
**CPU Core:** cortex-m4
**Chip name:** SAM4E Series
**SRAM size:** 128K bytes
**Flash size:** 1024K bytes

**Connections Info:**
**Board IP:** 192.168.0.100
**Your IP:** 192.168.0.1
**Total status requests:** 27

**FreeRTOS thread usage:**

| | |
|---|---|
| HTTP-req | <1% |
| IDLE | 90% |
| GFX | 1% |
| HTTP | <1% |
| GMAC | 1% |
| tcpip_thr | 6% |

**LwIP stats:**

| | | | | | |
|---|---|---|---|---|---|
| HEAP | Cur:0 | Size:14336 | Max:14160 | Err:0 | Usage:0% |
| UDP_PCB | Cur:0 | Size:4 | Max:0 | Err:0 | Usage:0% |
| TCP_PCB | Cur:16 | Size:16 | Max:16 | Err:0 | Usage:100% |
| TCP_PCB_LISTEN | Cur:1 | Size:1 | Max:1 | Err:0 | Usage:100% |
| TCP_SEG | Cur:0 | Size:16 | Max:12 | Err:0 | Usage:0% |
| REASSDATA | Cur:0 | Size:5 | Max:0 | Err:0 | Usage:0% |
| FRAG_PBUF | Cur:0 | Size:15 | Max:0 | Err:0 | Usage:0% |
| NETBUF | Cur:1 | Size:8 | Max:5 | Err:0 | Usage:13% |
| NETCONN | Cur:2 | Size:16 | Max:6 | Err:0 | Usage:13% |
| TCPIP_MSG_API | Cur:0 | Size:8 | Max:0 | Err:0 | Usage:0% |
| TCPIP_MSG_INPKT | Cur:0 | Size:8 | Max:3 | Err:0 | Usage:0% |
| SYS_TIMEOUT | Cur:5 | Size:5 | Max:5 | Err:0 | Usage:100% |
| PBUF_REF/ROM | Cur:0 | Size:10 | Max:0 | Err:0 | Usage:0% |
| PBUF_POOL | Cur:4 | Size:10 | Max:8 | Err:0 | Usage:40% |

**Transfer rate on ethernet MAC:**

### 5.2.1 Code Overview

As this demo is based on the Netconn API, the first step is to define the main tasks and to start the FreeRTOS scheduler. The following code is an extract of the thirdparty/lwip/netconn_http_stats_example/main.c file:

```
void main(void)
{
          /* Prepare the hardware to run this demo. */
          prvSetupHardware();
          …
          /** Create GFX task. */
          create_gfx_task(mainGFX_TASK_STACK_SIZE, mainGFX_TASK_PRIORITY);

          /** Create WebServer task. */
          create_http_task(mainHTTP_TASK_STACK_SIZE, mainHTTP_TASK_PRIORITY);

          /** Start the RTOS scheduler. */
          vTaskStartScheduler();
          …
}
```

The prvSetupHardware() function initializes the system clock to work at the maximum operating frequency, then it configures the required PIOs. Finally two tasks are created:

- GFX task performs the actual lwIP initialization based on user touchscreen input for IP address setting (fixed or DHCP). When init is complete, the assigned IP address is displayed and a blinking message let the user know that the device is ready to serve HTTP requests
- HTTP task waits for the GFX task for the user IP address input setting, then starts listening on the TCP port 80 (HTTP)

Priorities and stack sizes are defined in the thirdparty/lwip/netconn_http_stats_example/task.h file as following:

```
       #define mainGFX_TASK_PRIORITY         (tskIDLE_PRIORITY + 1)
       #define mainHTTP_TASK_PRIORITY        (tskIDLE_PRIORITY + 2)
```

The GFX task performs the ILI9325 touchscreen initialization then displays the IP address setting menu. The global variable *g_ip_mode* is updated depending on the user input (to select fixed IP address or DHCP mode) and the lwIP initialization is started calling the init_ethernet() function. This function is located in the thirdparty/lwip/netconn_http_stats_example/network/ethernet.c file.

```
void init_ethernet(void)
       {
          /* Initialize lwIP. */
          /* Call tcpip_init for threaded lwIP mode. */
          tcpip_init(NULL, NULL);

          /* Set hw and IP parameters, initialize MAC too */
          ethernet_configure_interface();
}
```

The tcpip_init() performs the actual initialization of the lwIP network stack in threaded mode; meaning that the lwIP threads are completely handled by the stack. The NULL parameters could be replaced with a function callback and arguments to know when the tcpip-thread (core lwIP thread) is up and running.

The ethernet_configure_interface() function is detailed below:

```
static void ethernet_configure_interface(void)
{
       …
       /* Add data to netif */
       /* Use ethernet_input as input method for standalone lwIP mode. */
       /* Use tcpip_input as input method for threaded lwIP mode. */
       If ( NULL == netif_add(&gs_net_if, &x_ip_addr, &x_net_mask, &x_gateway, NULL,
       ethernetif_init, tcpip_input) ) {
           …
       }

       /* Make it the default interface */
       netif_set_default(&gs_net_if);

       /* Setup callback function for netif status change */
       netif_set_status_callback(&gs_net_if, status_callback);

       /* Bring it up */
       if (g_ip_mode == 2) {
           /* DHCP mode. */
           if (ERR_OK != dhcp_start(&gs_net_if)) {
               …
           }
       }
       else {
           /* Static mode. */
           netif_set_up(&gs_net_if);
       }
}
```

Depending on the global variable *g_ip_mode* (user IP preference); dhcp is started or a static IP address (defined in the thirdparty/lwip/netconn_http_stats_example/sam4e16e_sam4e_ek/conf_eth.h file) is used.

A pointer to a status_callback() function is used to notify the GFX task that the lwIP configuration is done and a valid IP address can be displayed on screen.

The HTTP thread is in charge of serving HTTP requests. The main function of this task is defined in the thirdparty/lwip/netconn_http_stats_example/task_http.c file:

```
static void http_task(void *pvParameters)
{
       …
       /** Wait for user to read instructions. */
       WAIT_FOR_TOUCH_EVENT;
       …
       /* Create a new TCP connection handle */
       conn = netconn_new(NETCONN_TCP);
       …
       /* Bind to port 80 (HTTP) with default IP address */
       netconn_bind(conn, NULL, 80);

       /* Put the connection into LISTEN state */
       netconn_listen(conn);

       do {
           err = netconn_accept(conn, &newconn);
           if (err == ERR_OK) {
           /* Try to instanciate a new HTTP-req task to handle the HTTP request. */
               if (NULL == sys_thread_new("HTTP-req", http_request, newconn,
                       mainHTTP_TASK_STACK_SIZE, mainHTTP_TASK_PRIORITY)) {
                   /* Failed to instanciate task, free netconn socket. */
                   netconn_close(newconn);
                   netconn_delete(newconn);
               }
           }
       } while (err == ERR_OK);
       …
       /* Free netconn socket. */
       netconn_close(conn);
       netconn_delete(conn);

       /* Delete the calling task. */
       vTaskDelete(NULL);
}
```

The HTTP thread performs the following tasks:

- Wait for IP configuration and lwIP initialization
- Create a TCP socket using the Netconn API (NETCONN_TCP parameter)
- Bind the socket to TCP port 80 (HTTP)
- Put the socket into LISTEN state
- Accept input connections and instanciate one HTTP-req thread to handle the request
- In case of errors; close and delete the Netconn socket

By instanciating one HTTP-req thread per request; the main HTTP thread is always available to accept new requests, hence several HTTP clients can connect to the server at the same time.

HTTP-req threads are instanciated and use the http_request() function as the thread main function (source file is located in thirdparty/lwip/netconn_http_stats_example/network/httpserver/httpd.c).

```
void http_request(void *pvParameters)
{
        …
        /* Read the data from the port, blocking if nothing yet there. */
        /* We assume the request is in one netbuf. */
        if (ERR_OK == netconn_recv(conn, &inbuf)) {
            /* Read data from netbuf to the provided buffer. */
            netbuf_data(inbuf, (void**)&buf, &buflen);

            memset(req_string, 0, sizeof(req_string));
            http_getPageName(buf, buflen, req_string, sizeof(req_string));
            …
            /* Try to get a CGI handler for the request. */
            cgi = cgi_search(req_string, cgi_table);
            if (cgi) {
                /* Answer CGI request. */
                if (cgi(conn, req_string, buf, buflen) < 0) {
                    …
                }
            }
            /* Normal HTTP page request. */
            else {
                if (fs_open(req_string, &file) == 0) {
                    …
                }
                else {
                    /* Send the HTML header for file type. */
                    int type = http_searchContentType(req_string);
                    http_sendOk(conn, type);
                    …
                    netconn_write(conn, file.data, file.len, NETCONN_COPY);
                }
            }
        }

        /* Close the connection (server closes in HTTP). */
        netconn_close(conn);
        /* Delete the buffer (netconn_recv gives us ownership, */
        /* so we have to make sure to deallocate the buffer). */
        netbuf_delete(inbuf);

        /* Free resource. */
        netconn_delete(conn);

        /* Delete the calling task. */
        vTaskDelete(NULL);
}
```

The HTTP-req thread performs the following tasks:

- Read the data containing the HTTP request from the netconn socket to a netbuf structure
- Extract the actual request from the netbuf structure
- Handle CGI requests using helper functions
- Write to the netconn socket using netconn_write() function calls to answer the request
- Close the connection
- Free netbuf and netconn socket resources
- Free FreeRTOS resource by deleting the current HTTP-req task

### 5.2.2 Memory Footprint

The memory footprint information in Table 5-2 has been obtained using IAR 6.50.5 compiler with high optimization for size.

**Table 5-2. HTTP Netconn Example Memory Footprint**

| Modules | Flash (bytes) | | SRAM (bytes) |
|---|---|---|---|
| | RO code | RO data | RW data |
| GMAC+PHY driver | 2180 | 0 | 4648 |
| lwIP stack | 27980 | 30 | 35711 |
| FreeRTOS | 1768 | 9 | 20564 |
| SAM4E other drivers | 6094 | 250 | 1172 |
| User application | 7416 | 430639 | 2080 |
| Total | 45440 | 430928 | 64175 |
| Others (libc, stack, etc) | 6400 | 50 | 1024 |
| Grand Total | 51840 | 430978 | 65199 |

The following memory configuration was used for the lwIP network stack:

- 3 TX buffers of 1536 bytes for the GMAC driver (conf_eth.h)
- 10 buffers of 1536 bytes for the lwIP buffer pool (lwipopts.h)
- 14K for the lwIP heap memory (lwipopts.h)

The following memory configuration was used for FreeRTOS:

- 20K for the FreeRTOS heap memory (FreeRTOSConfig.h)

Note that this memory footprint information is not optimal and can be reduced depending on your requirements.

# 6. Debugging with lwIP

The lwIP network stack integrates some nice support for enabling various debug print options. The debug configuration is done in the debug section at the end of the lwipopts.h file.

Here are the recommanded steps to enable debug output:

- Define *LWIP_DEBUG* to enable print output
  It simply redirects *LWIP_PLATFORM_ASSERT and LWIP_PLATFORM_DIAG* to printf() function (as defined in the thirdparty/lwip/lwip-port-1.4.1/sam/include/arch/cc.h file).
- Define *LWIP_DBG_MIN_LEVEL* to the desired level of debug output
  The possible values are listed in the thirdparty/lwip/lwip-1.4.1/src/include/lwip/debug.h file.
- Define *LWIP_DBG_TYPES_ON* to *LWIP_DBG_ON* to enable the use of various *xxx_DEBUG* options
  This define is actually used as a quicker way to disable *LWIP_DEBUGF* messages all at once.
- Define any desired *xxx_DEBUG* option to *LWIP_DBG_ON* to enable the corresponding debug output
  Simply define to *LWIP_DBG_OFF* to turn the corresponding *xxx_DEBUG* option off.
- Comment the *LWIP_NOASSERT* define to enable assertion tests
  Assert will only loop without printing a message if *LWIP_DEBUG* is undefined.

The following configuration can be used to print debug messages from the network interface driver:

```
#define LWIP_DEBUG              1
#define LWIP_DBG_MIN_LEVEL      LWIP_DBG_LEVEL_ALL
#define LWIP_DBG_TYPES_ON       LWIP_DBG_ON
#define NETIF_DEBUG             LWIP_DBG_ON
```

Note: Your terminal must be configured to treat LF characters as new-line to properly print debug messages.

# 7. Optimizing the SRAM Footprint

## 7.1 Heap Memory

The lwIP heap memory is defined in the thirdparty/lwip/lwip -1.4.1/src/core/mem.c file as a byte array called "ram_heap" with a size of *MEM_SIZE*. The heap memory is used for dynamic memory allocation. When the user application sends data to a remote host, the network stack has to build the corresponding Ethernet frame meaning that a TCP header, an IP header and an Ethernet header will be allocated and prepended to the pbuf chain. Depending on the amount of data to send, the send buffer size and the send queue length configuration, the heap memory size *MEM_SIZE* can be hard to adjust.

The lwIP network stack provides two different approaches to find heap memory allocation failures at runtime:

- By using the memory debug messages. Set the following defines in the lwIP configuration file:

  ```
  #define LWIP_DEBUG                  1
  #define MEM_DEBUG                   LWIP_DBG_ON
  ```

  Then run the application; perform network operations (http page request etc). Memory allocation failures will be directly printed on the UART console.

- By using the lwIP stats system. Set the following defines in the lwIP configuration file:

  ```
  #define LWIP_STATS                  1
  #define MEM_STATS                   LWIP_DBG_ON
  #define MEMP_STATS                  1
  #define MEM_DEBUG                   LWIP_DBG_ON
  ```

  Then run the application; perform network operations (http page request etc), break the program and watch the "lwip_stats" array in the thirdparty/lwip/lwip -1.4.1/src/core/lwip/stats.c file. The inner mem structure represents the heap memory; if error count is superior to zero, some malloc calls have failed.

Memory heap allocation errors are not necessarily critical if handled properly by the user application. However, memory heap allocation errors usually have a bad influence in terms of network latency and throughput.

## 7.2 Memory Pools

The lwIP network stack uses memory pools to fasten the memory allocation process. The various pools are defined in the thirdparty/lwip/lwip -1.4.1/src/include/lwip/memp_std.h file and each pool size can be configured in the lwipopts.h file.

The best approach to properly adjust a pool size is by enabling the stats system in the lwIP configuration file:

```
#define LWIP_STATS                  1
#define MEMP_STATS                  1
```

Also enable the debug output to read the pool name field:

```
#define LWIP_DEBUG                  1
#define LWIP_DBG_MIN_LEVEL          LWIP_DBG_LEVEL_ALL
#define LWIP_DBG_TYPES_ON           LWIP_DBG_ON
#define MEMP_DEBUG                  LWIP_DBG_ON
```

Then run the application; perform network operations (http page request etc) and stop the debugger. Watch the "lwip_stats" array: the "memp" element (array) contains all the memory pools information. Each pool is listing with the following fields:

- **name**: the name of the pool
- **avail**: the number of currently free elements in the pool
- **used**: the number of currently used elements in the pool
- **max**: the highest number of used elements in the pool since startup time
- **err**: the number of allocation error (out of memory) in the pool

The size of each element in a pool can be seen at runtime by watching the "memp_sizes" array.

The best approach to optimize the memory pool footprint is to adjust the various pool configuration values in the lwIP configuration file so that the max field equals the avail field. Unused pools can also be disabled in this configuration file.

## 7.3 Stack Size

If no RTOS is used:

In IAR the stack is defined by the CSTACK section. It appears like this in the map file:

```
    CSTACK          uninit   0x20003f48   0x100  <Block tail>
                         - 0x20004048   0x100
```

Note: The stack is configured to grow up.

Optimizing the stack size is possible by choosing the lowest possible stack size without risking a stack overflow. To find the best value, place a breakpoint at the beginning of the main function. Run the program to the breakpoint. Open the memory view, go the CSTACK block tail, and write a memory pattern (like 0xDEADBEEF) until the stack start address. Release the application from the breakpoint, fully test the program features, stop the program and inspect memory view at CSTACK block tail address. If the memory pattern is still widely present, the stack size can be safely decreased. However, if the memory pattern is gone, the stack size is too short and must be increased. It is recommended to keep a security margin in the stack size to handle any situation.

If RTOS is used:

Refer to the RTOS manual for stack debugging.

## 8. Revision History

| Doc. Rev. | Date | Comments |
|---|---|---|
| 42233A | 03/2014 | Initial document release |

Atmel | Enabling Unlimited Possibilities®

**Atmel Corporation**
1600 Technology Drive
San Jose, CA 95110
USA
**Tel:** (+1)(408) 441-0311
**Fax:** (+1)(408) 487-2600
www.atmel.com

**Atmel Asia Limited**
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
**Tel:** (+852) 2245-6100
**Fax:** (+852) 2722-1369

**Atmel Munich GmbH**
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
**Tel:** (+49) 89-31970-0
**Fax:** (+49) 89-3194621

**Atmel Japan G.K.**
16F Shin-Osaki Kangyo Building
1-6-4 Osaki, Shinagawa-ku
Tokyo 141-0032
JAPAN
**Tel:** (+81)(3) 6417-0300
**Fax:** (+81)(3) 6417-0370