Explorando o stack lwIP

Ronaldo Duarte

Quem sou eu?

- Engenheiro de computação (UNICAMP, 2002)
- 10 anos de experiência em sistemas embarcados

Agenda

1. Introdução
2. Arquitetura do lwIP
3. Modos de operação
4. Driver
5. Packet buffers
6. Módulos principais
7. API baixo nível (usando UDP e TCP)
8. Sockets BSD
9. Footprint
10. Demonstração

Introdução

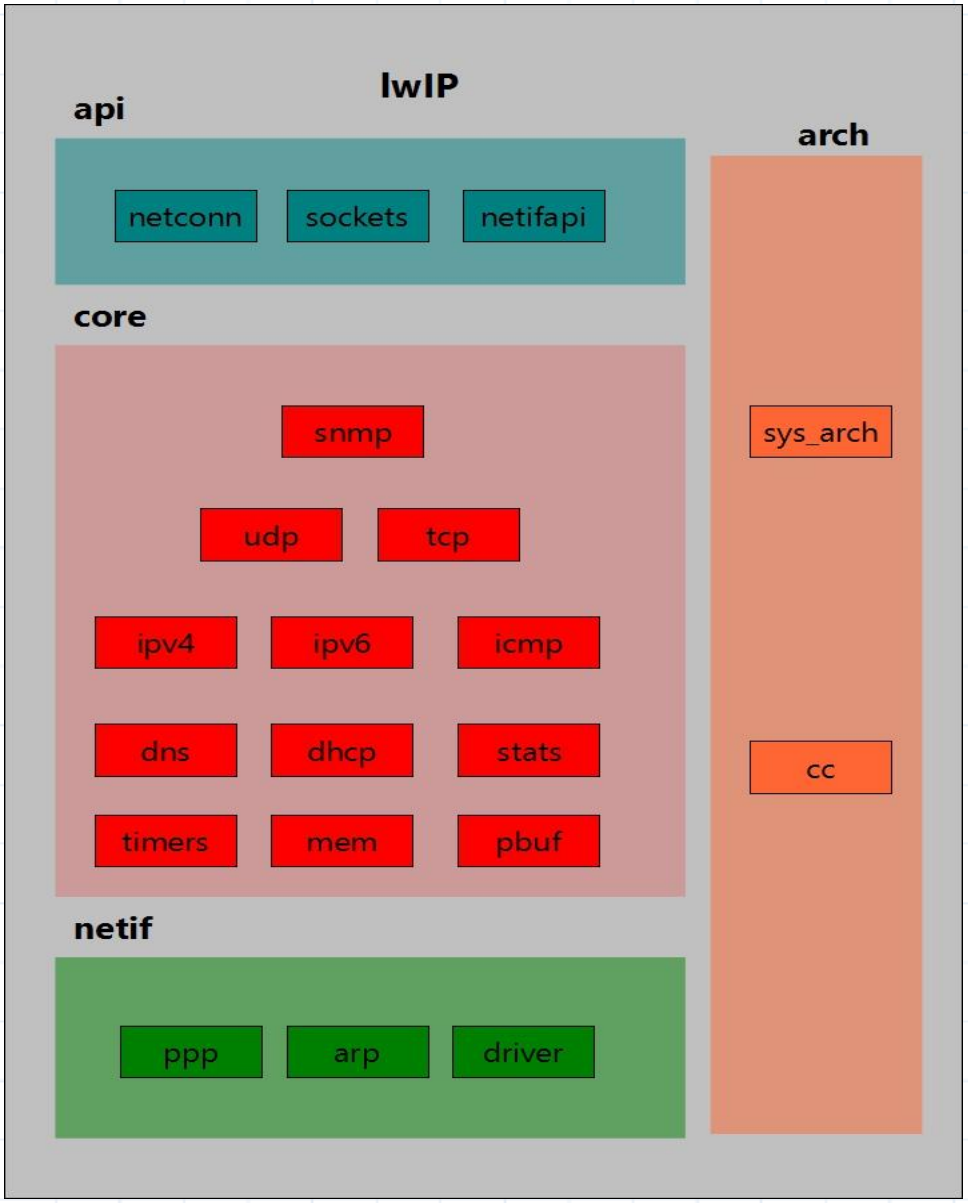
- **l**ight**w**eight **IP**
- Pilha de protocolos TCP/IP open source
- Desenvolvido inicialmente por Adam Dunkels (SICS)
- Foco em sistemas embarcados (recursos limitados)
- Licença BSD modificada
- Opção ainda mais enxuta: uIP

Introdução

- Aplicações interessantes:
  - Analog Devices (Blackfin)
  - Altera (Nios II)

- Protocolos inclusos:
  - ARP, PPP, DHCP
  - IP, ICMP, IGMP
  - TCP, UDP
  - SNMP, DNS

Arquitetura



- Módulos organizados em camadas espelhando a pilha TCP/IP
- Violações de encapsulamento com fins de desempenho, ex: buffers

Customização

- Arquivo *opt.h* com definições padrões
    - mais de 200 parâmetros

- Arquivo *lwipopts.h* com parâmetros ajustados para a aplicação

- Flexibilidade enorme de uso de recursos

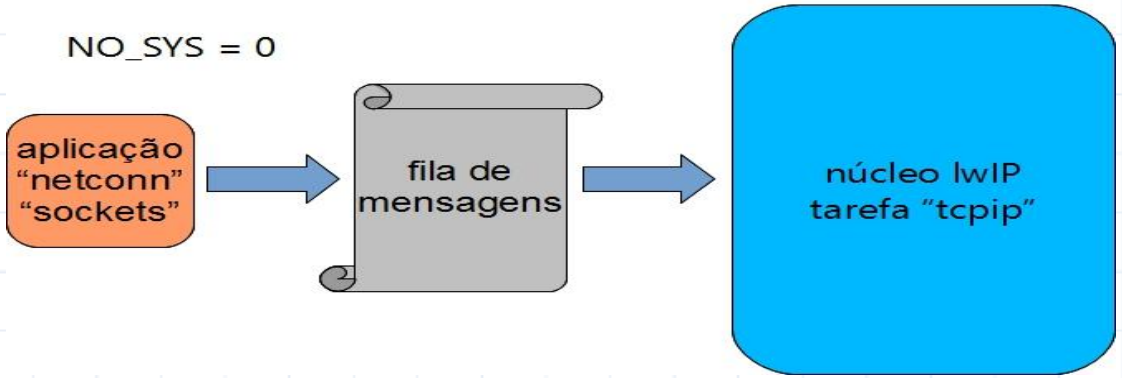
Modos de operação

Aplicação e núcleo lwIP na mesma tarefa?

NO_SYS = 1

"raw" : API baixo nível baseada em eventos

NO_SYS = 0



"netconn" e "sockets": APIs sequenciais

Abstração de OS

- lwIP portável => camada de emulação de OS

- interface uniforme para acesso a serviços de OS (criação de tarefas, primitivas de sincronização, passagem de mensagens)

```
sys_thread_t sys_thread_new( const char *pcName, void( *pxThread )( void *pvParameters ), void *pvArg, int iStackSize, int iPriority )
{
xTaskHandle xCreatedTask;
portBASE_TYPE xResult;
sys_thread_t xReturn;

    xResult = xTaskCreate( pxThread, ( signed char * ) pcName, iStackSize, pvArg, iPriority, &xCreatedTask );

    if( xResult == pdPASS )
    {
        xReturn = xCreatedTask;
    }
    else
    {
        xReturn = NULL;
    }

    return xReturn;
}
```

Driver

- Drivers para hardware de rede representados por interfaces de rede (netif)
- Abstraem para o lwIP a recepção e envio de frames
- Várias interfaces operando simultaneamente

```
struct netif *
netif_add(struct netif *netif, ip_addr_t *ipaddr, ip_addr_t *netmask,
  ip_addr_t *gw, void *state, netif_init_fn init, netif_input_fn input)
```

```
/** Function prototype for netif->input functions. This function is saved as 'input'
 * callback function in the netif struct. Call it when a packet has been received.
 *
 * @param p The received packet, copied into a pbuf
 * @param inp The netif which received the packet
 */
typedef err_t (*netif_input_fn)(struct pbuf *p, struct netif *inp);
```

```
/** Function prototype for netif->linkoutput functions. Only used for ethernet
 * netifs. This function is called by ARP when a packet shall be sent.
 *
 * @param netif The netif which shall send a packet
 * @param p The packet to send (raw ethernet packet)
 */
typedef err_t (*netif_linkoutput_fn)(struct netif *netif, struct pbuf *p);
```

Driver

- Controlador Ethernet com DMA

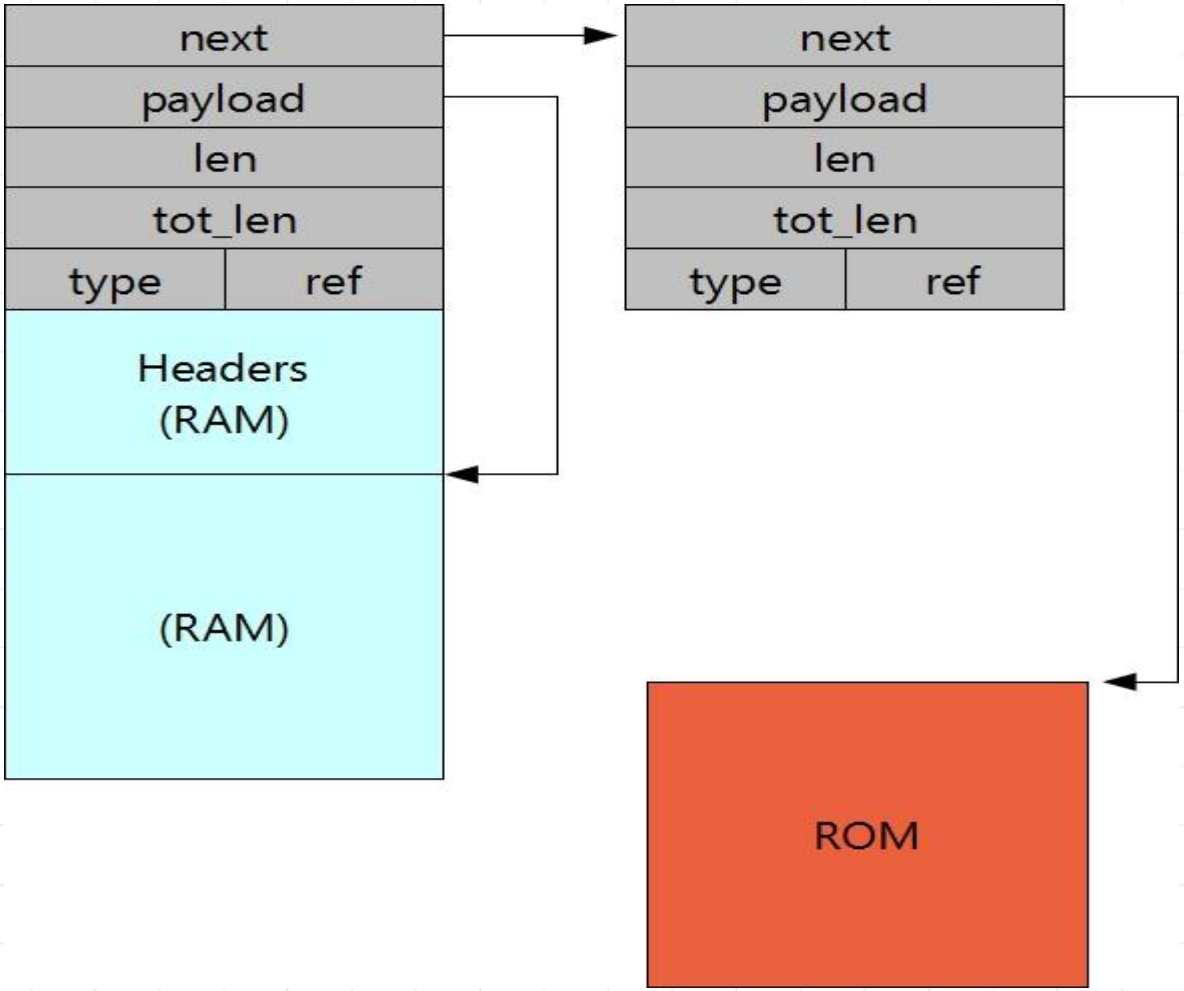
EVITAR CÓPIAS!

Buffers

- Eficientes para lidar com tamanhos diferentes
- Preparados para evitar cópias (vilão memcpy!)
- Solução: pbufs (packet buffers)
- Preparados para alocar memória dinâmica ou trabalhar com memória estática pré-alocada.
- Podem ser encadeados (pbuf chain)

pbufs: tipos

- PBUF_RAM, PBUF_ROM, PBUF_REF e PBUF_POOL



PBUF_POOL: recepção de dados no driver (alocação rápida)

PBUF_RAM, PBUF_REF, PBUF_ROM: envio de dados da aplicação (flexível)

pbufs: configuração

- **PBUF_POOL_SIZE**
  Número de pbufs no pool

- **PBUF_POOL_BUFSIZE**
  Tamanho de cada pbuf do pool

pbufs: memória

- pbufs do tipo PBUF_POOL obtém memória de um pool de pbufs previamente alocado

- pbufs do tipo PBUF_RAM obtém memória dinâmica através de um alocador simples (blocos) com prevenção de fragmentação

- pbufs do tipo PBUF_REF e PBUF_ROM apenas referenciam memória previamente alocada

ARP

- Manutenção automática da tabela ARP
- Ponto de entrada no lwIP para redes com hardware Ethernet
- pbuf (possivelmente uma cadeia) alocado pelo driver e entregue ao lwIP

```
err_t
ethernet_input(struct pbuf *p, struct netif *netif)
```

IP

- IPv4 e IPv6
- Envio de pacotes IP após identificar uma rota através do endereço IP e máscara de uma interface (netif).
- Fragmentação IP tratada (pacotes que excedem o MTU)
- ICMP tratado automaticamente (“ping”)

SNMP

- Protocolo de gerenciamento de redes
- lwIP pronto para coletar estatísticas descritas no RFC 1213 (MIB-II)
- Opção **LWIP_SNMP**

Interface com aplicação

- 3 APIs disponíveis:
  - baixo nível, baseada em callbacks (“raw”)
  - alto nível, sequencial (“netconn”)
  - sockets BSD!
- Escolha baseada no compromisso:
  - velocidade de execução
  - facilidade
  - comprometimento de memória

API baixo nível

- Aplicação e núcleo lwIP na mesma tarefa
- Única opção para sistemas sem OS
- Execução movida por callbacks
- Acesso aos serviços TCP e UDP
- Estado mantido por PCBs (protocol control block)

Conexão TCP

- Passos para abertura passiva de conexão

1. Obter um PCB: `struct tcp_pcb *tcp_new(void)`

2. "bind": `err_t tcp_bind(struct tcp_pcb *pcb, ip_addr_t *ipaddr, u16_t port)`
   - Definição de endereço local e porta
   - IP_ADDR_ANY para todos endereços locais

3. "listen": `struct tcp_pcb *tcp_listen(struct tcp_pcb *pcb)`

4. "accept": define callback para conexão aceita

```
void tcp_accept(struct tcp_pcb *pcb, err_t (* accept)(void *arg, struct tcp_pcb *newpcb, err_t err))
```

Conexão TCP

5. Callback fornece o PCB conectado (diferente do PCB utilizado no “listen”)

6. Informar que uma conexão foi aceita

```
void tcp_accepted(struct tcp_pcb *pcb)
```

Conexão TCP

- Passos para abertura ativa de conexão

1. Obter um PCB: `struct tcp_pcb *tcp_new(void)`
2. "connect": produz um segmento SYN

```
err_t tcp_connect(struct tcp_pcb *pcb, ip_addr_t *ipaddr, u16_t port, err_t (* connected)(void *arg, struct tcp_pcb *tpcb, err_t err));
```

3. Aviso de conexão estabelecida

Conexão TCP

- Envio de dados em conexão TCP:

1. Informar o callback invocado quando o host remoto reconhecer os dados (flag ACK)

```
void tcp_sent(struct tcp_pcb *pcb, err_t (* sent)(void *arg, struct tcp_pcb *tpcb, u16_t len))
```

2. Usar o PCB conectado em tcp_write( )

```
err_t tcp_write(struct tcp_pcb *pcb, const void *dataptr, u16_t len, u8_t apiflags)
```

Conexão TCP

- Recepção de dados em conexão TCP:

1. Informar o callback invocado quando dados forem recebidos na conexão

```
void tcp_recv(struct tcp_pcb *pcb, err_t (* recv)(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err))
```

2. Após tratar os dados no callback, informar que a aplicação consumiu os dados para que o TCP anuncie uma nova janela para o par

```
void tcp_recved(struct tcp_pcb *pcb, u16_t len)
```

Conexão TCP

- Finalização da conexão:

tcp_close( ) no PCB conectado

```
err_t tcp_close(struct tcp_pcb *pcb)
```

Comunicação UDP

• UDP consideravelmente mais simples que TCP, simplicidade refletida na API. Passos:

1. Obter um PCB: `struct udp_pcb *udp_new(void)`

2. "bind": `err_t udp_bind(struct udp_pcb *pcb, ip_addr_t *ipaddr, u16_t port)`
   - Definição de endereço local e porta
   - IP_ADDR_ANY para todos endereços locais

3. "connect": configura o endereço e porta do host remoto, não gera tráfego de rede

```
err_t udp_connect(struct udp_pcb *pcb, ip_addr_t *ipaddr, u16_t port)
```

Comunicação UDP

4. Envio:

```
err_t udp_send(struct udp_pcb *pcb, struct pbuf *p)
```

5. Callback para recepção:

```
void udp_recv(struct udp_pcb *pcb, void (* recv)(void *arg, struct udp_pcb *upcb, struct pbuf *p, ip_addr_t *addr, u16_t port), void *recv_arg)
```

APIs sequenciais

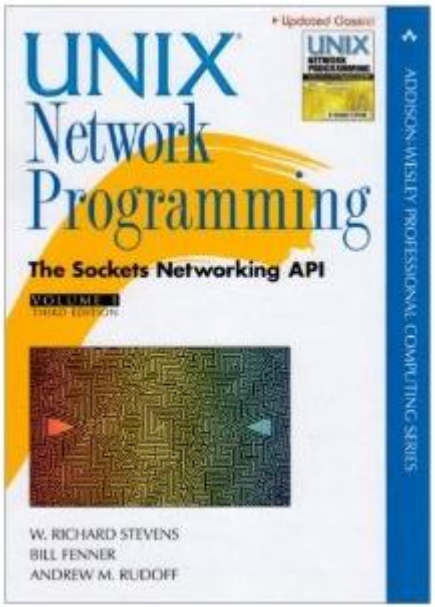
- Exigem um OS
- Comunicação com núcleo lwIP por mensagens
- Sequenciais, mais produtivas, menos eficientes
- Disponíveis:
  - netconn
  - sockets BSD

APIs sequenciais

- netconn: utiliza a API baixo nível para troca de dados sequenciais
- sockets BSD: fina camada sobre a API netconn

Sockets BSD

- Berkeley sockets
- API padrão (POSIX) em todos os OS modernos
- Desenvolvedores produtivos no lwIP imediatamente!



Sockets BSD

- **LWIP_COMPAT_SOCKETS**

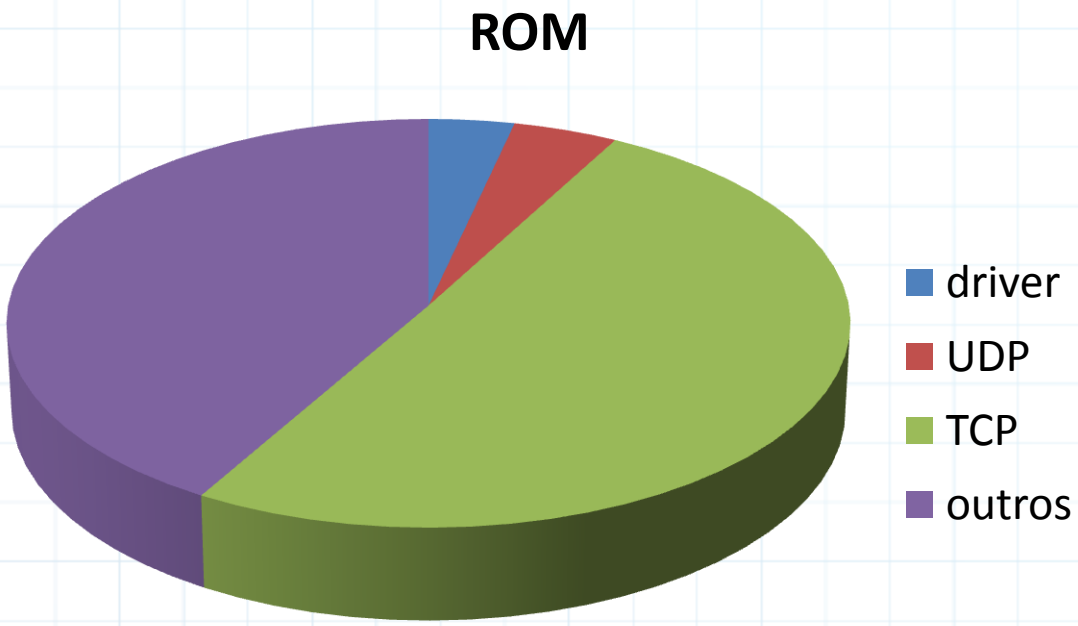
```
#if LWIP_COMPAT_SOCKETS
#define accept(a,b,c)         lwip_accept(a,b,c)
#define bind(a,b,c)           lwip_bind(a,b,c)
#define shutdown(a,b)         lwip_shutdown(a,b)
#define closesocket(s)        lwip_close(s)
#define connect(a,b,c)        lwip_connect(a,b,c)
#define getsockname(a,b,c)    lwip_getsockname(a,b,c)
#define getpeername(a,b,c)    lwip_getpeername(a,b,c)
#define setsockopt(a,b,c,d,e) lwip_setsockopt(a,b,c,d,e)
#define getsockopt(a,b,c,d,e) lwip_getsockopt(a,b,c,d,e)
#define listen(a,b)           lwip_listen(a,b)
#define recv(a,b,c,d)         lwip_recv(a,b,c,d)
#define recvfrom(a,b,c,d,e,f) lwip_recvfrom(a,b,c,d,e,f)
#define send(a,b,c,d)         lwip_send(a,b,c,d)
#define sendto(a,b,c,d,e,f)   lwip_sendto(a,b,c,d,e,f)
#define socket(a,b,c)         lwip_socket(a,b,c)
#define select(a,b,c,d,e)     lwip_select(a,b,c,d,e)
#define ioctlsocket(a,b,c)    lwip_ioctl(a,b,c)
```

Sockets BSD

- Usar ou não?
- Portável, funcional e dominado pelos desenvolvedores.
- Porém:
  - API mais recente e menos usada no lwIP
  - mantenedores do projeto dão mais atenção ao núcleo
  - caso pessoal: select( )

Footprint (demonstração)

- 25.2 kB (ROM)
  - driver 0.9 kB
  - UDP 1.1 kB
  - TCP 12.7 kB



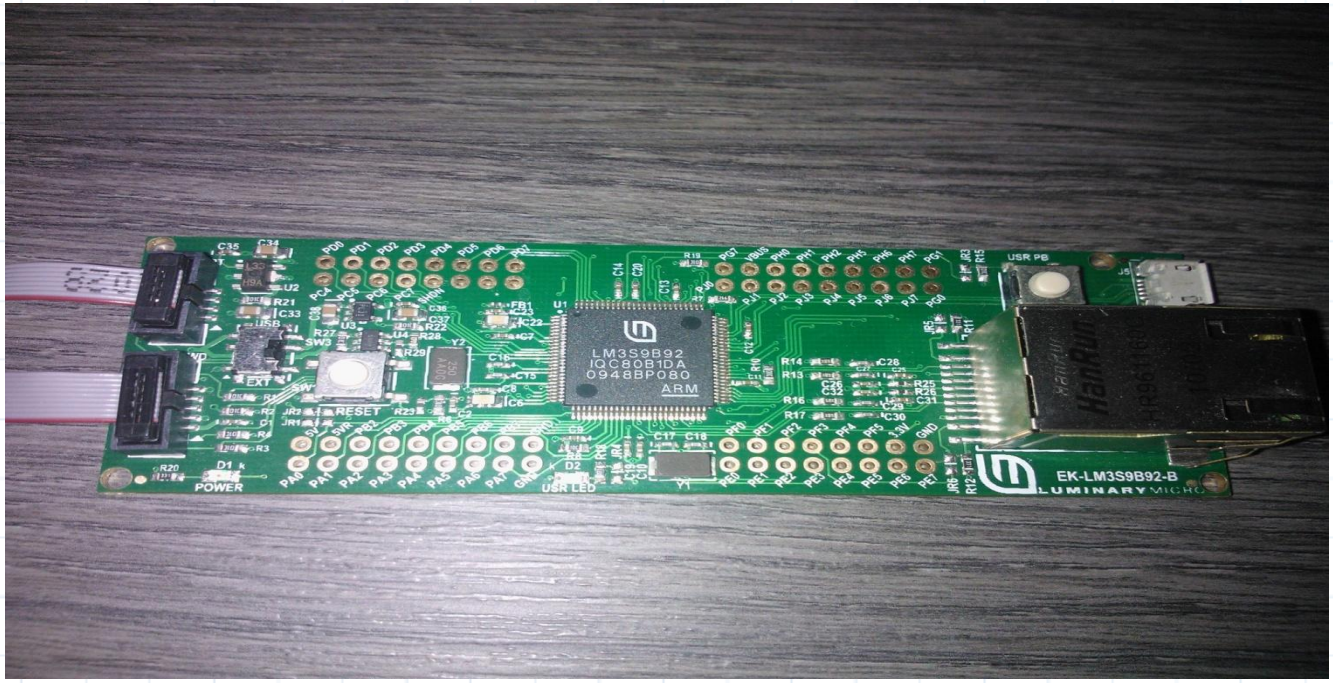
- 12.6 kB (RAM)
  - 1 PCB UDP
  - 1 PCB TCP
  - 16 pbufs no pool

Demonstração

Aplicação simples usando a API baixo nível

Cortex-M3 com MAC+PHY integrados

ARP + ICMP + UDP



Como continuo?

- savannah.nongnu.org/projects/lwip
- lwip.wikia.com/wiki/LwIP_Wiki
- lists.nongnu.org/mailman/listinfo/lwip-users
- lists.nongnu.org/mailman/listinfo/lwip-devel

Obrigado!

Ronaldo Duarte

ronaldo.tomazeli@gmail.com

@ronaldotd