LwIP : Porting for bare Metal

Steps:

1. **Create cc.h**

This file contains settings needed to adapt lwIP for your compiler and machine architecture. Rather than duplicate the description of this file

This is a basic header that describes your compiler and processor to lwIP. The following things must be defined:

- Data types:
    - `u8_t`, `u16_t`, `u32_t`: The unsigned data types. For a 32-bit microprocessor, this is usually `unsigned char`, `unsigned short`, and `unsigned int`.
    - `s8_t`, `s16_t`, `s32_t`: The signed counterparts.
    - `mem_ptr_t`: A generic pointer type. It has to be an integer type (not void*, due to some pointer arithmetics).

- printf formatters for data types:
    - U16_F, S16_F, X16_F, U32_F, S32_F, X32_F, SZT_F
    - usually defined to "hu", "d", "hx", "u", "d", "x", "uz"
    - These formatters are used in several lwip files, yet mainly for diagnostics (LWIP_PLATFORM_DIAG) and debug output (LWIP_DEBUGF), so you might not need them if you disable that output.

- Byte ordering aka endianness:

  #define BYTE_ORDER LITTLE_ENDIAN
  or
  #define BYTE_ORDER BIG_ENDIAN

Packet headers data is stored in network byte order which is the big-endian mode. If your processor architecture is little-endian, then we need to convert data using htons()/htonl()/ntohs()/ntohl() functions.

If your processor supports both big-endian and little-endian mode or you are yet free to chose a processor, big-endian might be a better choice regarding performance, as host- and network-byte order are the same.

LwIP provides standard functions for this, but it's often more efficient to define your own macros or functions if your compiler provides intrinsics or assembly instructions for byte swapping.

For example:

```
#define LWIP_PLATFORM_BYTESWAP 1
#define LWIP_PLATFORM_HTONS(x) ( (((u16_t)(x))>>8) |
(((x)&0xFF)<<8) )
#define LWIP_PLATFORM_HTONL(x) ( (((u32_t)(x))>>24) |
(((x)&0xFF0000)>>8) \
                        | (((x)&0xFF00)<<8) | (((x)&0xFF)<<24) )
```

- Computing cheksums:

  IP protocols use checksums (see RFC 1071). LwIP gives you a choice of 3 algorithms:

  1. load byte by byte, construct 16 bits word and add: not efficient for most platforms
  2. load first byte if odd address, loop processing 16 bits words, add last byte.
  3. load first byte and word if not 4 byte aligned, loop processing 32 bits words, add last word/byte.

  ```
  #define LWIP_CHKSUM_ALGORITHM 2
  ```

you may also use your own checksum routine (e.g. using assembly to add 32 bits words with carry)

```
u16_t my_chksum(void *dataptr, u16_t len);
#define LWIP_CHKSUM my_chksum
```

- Structure packing:

  LwIP accesses to 16 and 32 bits data fields in protocol headers which may be not aligned in memory. If your processor cannot read from/write to misaligned addresses, then you need to tell your compiler that the data may be not aligned and it must generate multiple byte or word load/stores to access it.

  In all protocol structures, 16/32 bits values are 16 bits aligned, so choosing a 2 byte alignment for structures should be safe.

  The common case is using Ethernet interfaces and MEM_ALIGNMENT=4. Ethernet header without VLAN is 14 bytes, so if you can/must have ETH_PAD_SIZE=2, then IP headers and higher layers are 4 bytes aligned and you may not need packing !

  In any case, make sure the system runs stable when chosing a structure packing setting different to 1!

  Usually a packed structure is created like that:

```
<   #ifdef PACK_STRUCT_USE_INCLUDES
  #  include "arch/bpstruct.h"
  #endif
  PACK_STRUCT_BEGIN
  struct <structure_name> {
     PACK_STRUCT_FIELD(<type> <field>);
     PACK_STRUCT_FIELD(<type> <field>);
```

```
  <...>
} PACK_STRUCT_STRUCT;
PACK_STRUCT_END
#ifdef PACK_STRUCT_USE_INCLUDES
#  include "arch/epstruct.h"
#endif
```

> Appropriate definitions of PACK_STRUCT_BEGIN, PACK_STRUCT_FIELD, PACK_STRUCT_STRUCT and PACK_STRUCT_END should be included in cc.h. The required definitions mainly depend on your compiler. Only PACK_STRUCT_STRUCT is required, lwip/src/include/lwip/arch.h provides empty defaults for the others. Optionally you can also define PACK_STRUCT_USE_INCLUDES and provide bpstruct.h and epstruct.h.

For example, according to a posting on the mailing list, the following values work for GCC:

```
#define PACK_STRUCT_FIELD(x) x __attribute__((packed))
#define PACK_STRUCT_STRUCT __attribute__((packed))
#define PACK_STRUCT_BEGIN
#define PACK_STRUCT_END
```

> Warning: In version 1.3.0 of lwip the following source files do not conform to this pattern::* lwip/src/netif/ppp/vjbsdhdr.h only uses PACK_STRUCT_BEGIN and PACK_STRUCT_END because it uses bitfields

- Platform specific diagnostic output
    - `LWIP_PLATFORM_DIAG(x)` - non-fatal, print a message. Uses printf formating.
    - `LWIP_PLATFORM_ASSERT(x)` - fatal, print message and abandon execution. Uses printf formating. Unlike assert() from the standard c library, the parameter x is the message, not a condition.

2. Create sys_arch.h

For the NO_SYS environment, no operating system adaptation layer is required, so this file merely contains a handful of typedefs and preprocessor definitions.

The sys_arch provides semaphores and mailboxes to lwIP. For the full lwIP functionality, multiple threads support can be implemented in the sys_arch, but this is not required for the basic lwIP functionality. Previous versions of lwIP required the sys_arch to implement timer scheduling as well but as of lwIP 0.5 this is implemented in a higher layer.

3. Create the lwipopts.h for your port.

You will probably want to find an existing version of this file from the lwIP contrib directory, copy it and then start modifying. The number one setting is to define the preprocessor symbol NO_SYS to 1.

There are a multitude of settings that can be specified in this file that determine which optional portions of lwIP will be included (*e.g.* IP reassembly, handling of out-of-order TCP segments, and so forth), which protocols will be enabled (TCP, UDP, ARP, DHCP, etc.), and a number of settings that help to determine how much memory will be used by the stack. The stack's source code is organized so that your settings in this file will override the built-in defaults in lwIP's lwip/opts.h file.

4. Creating network driver.

This process is the same whether or not you are using an operating system. See Writing a device driver for more info.

Figure 2-1. lwIP Receive Flow

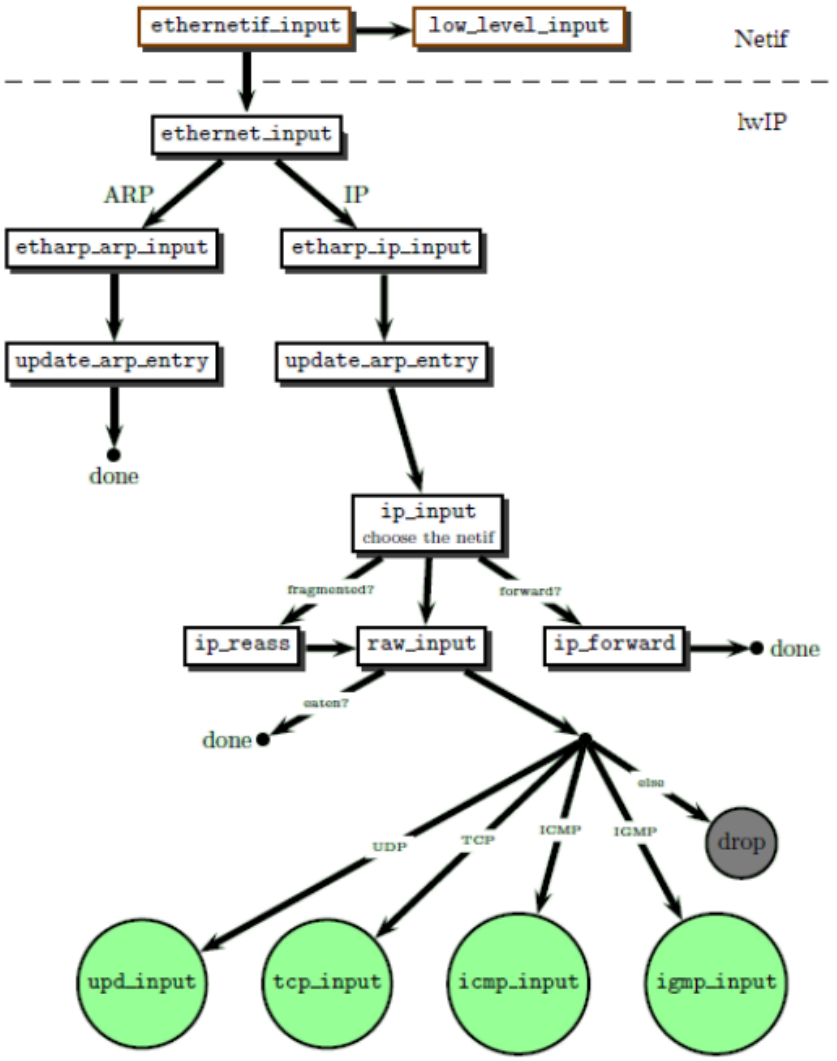
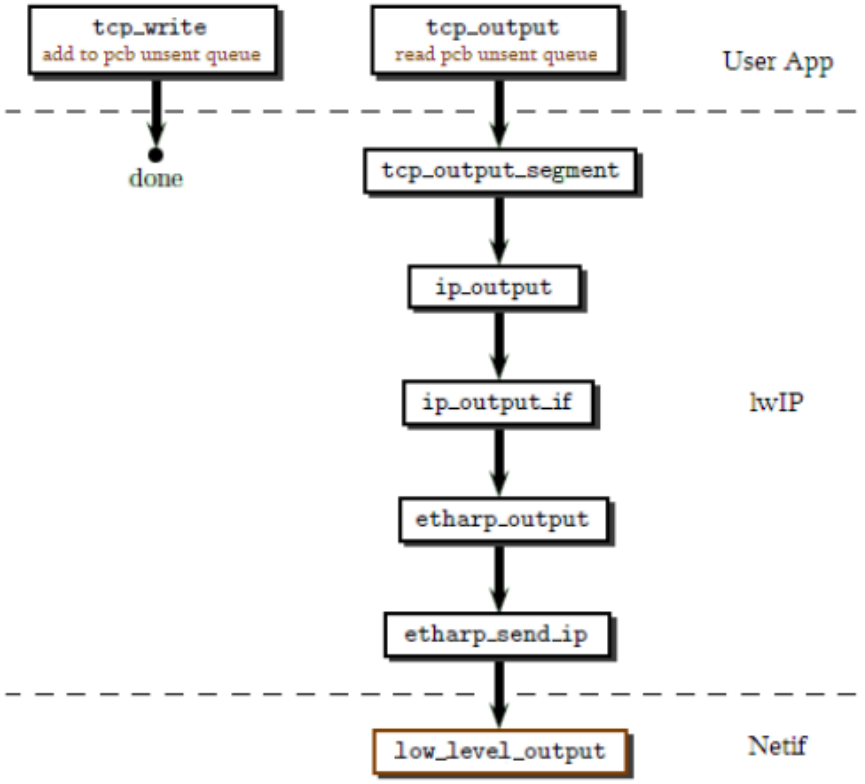


Figure 2-3. lwIP TCP Output Flow



5. Run “ping sender” test on AJIT, to verify above stuff works.

This is an example of a "ping" sender (with raw API and socket API).

It can be used as a start point to maintain opened a network connection, or

like a network "watchdog" for your device.

AI_NIC_DEMO/lwip/lwip-STABLE-2_2_0_RELEASE/contrib/apps/ping/ping.c

6. Ping AJIT from PC

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.
- timers_update() to refresh and trigger the lwIP timers.