cIPS

0

http://sourceforge.net/projects/cipsuite/

1. Abstract

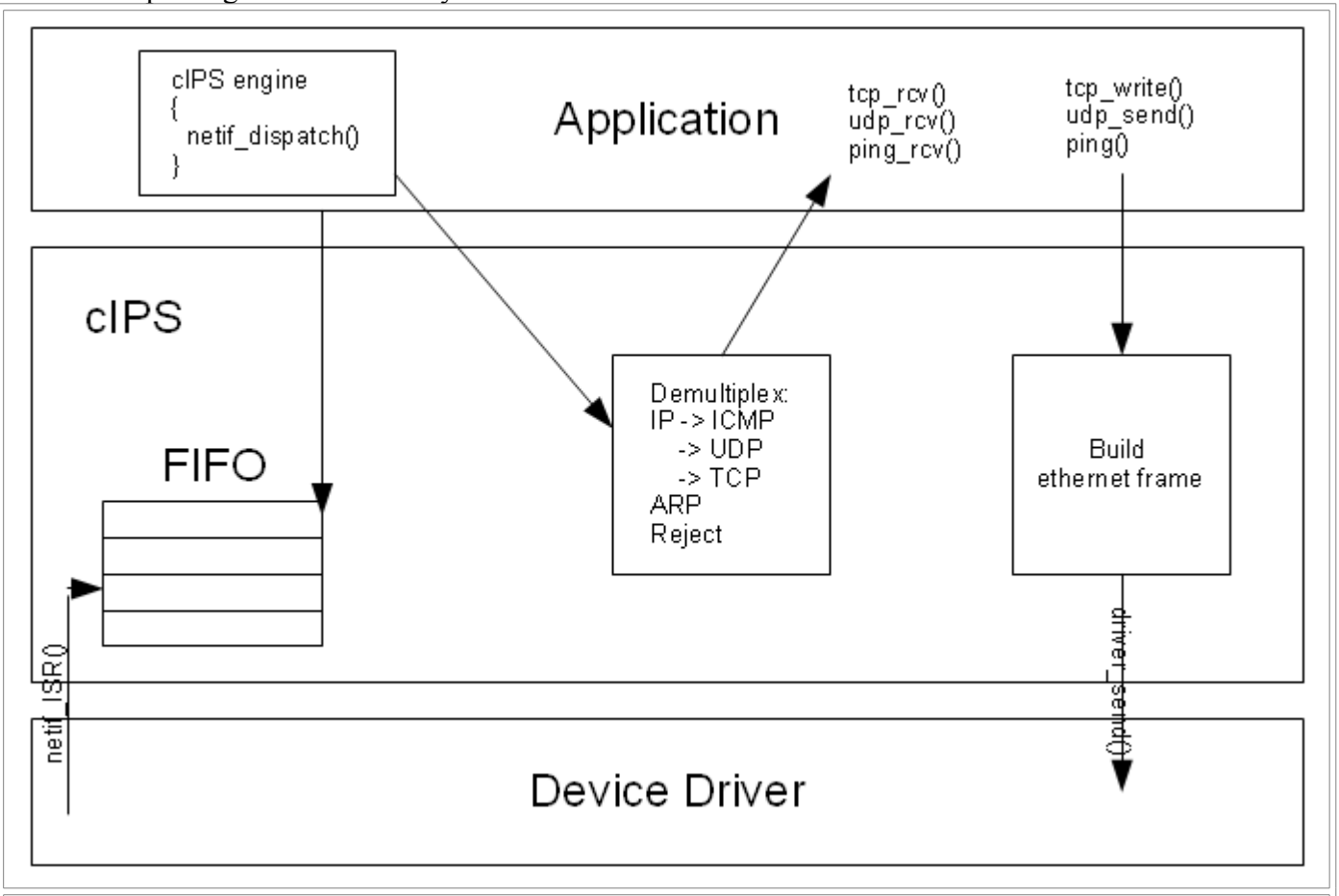
- cIPS stands for "compact Internet Protocol Suite" or "IP stack in C language". I was looking for a short meaningful name.
- **cIPS is a robust and fast TCP-IP library for embedded applications without an Operating System.**
  The motivation of this project was to implement a fast and reliable TCP-IP stack that could handle several network adapters on an embedded system.
- cIPS is entirely abstract from any platform and therefore does not contain any device drivers. It is some pure C code (to convince yourself, open the dev-C++ project provided or use the Makefile to compile).

2. Features

- Supports ARP, IP (v4), TCP and a subset of ICMP (ping). A subset of HTTP is implemented in a web server example but it is outside the library.
- Supports several network adapters.
- The maximum number of connections is configurable.
- The traces are readable and are an option. However, there is also an error report strategy (see 3.2 Error reporting)
- Data are separated from the processing.
- Does not perform any dynamic memory allocation.
- Fully tested on a big-endian platform. However swapping macros for little-endian platforms are presents.

3. Principle of the cIPS library

When a Ethernet frame comes, the device driver triggers an ISR called netif_ISR(). To keep that ISR short, it places the frame into a pool of frames. After, netif_dispatch() is in charge of scanning the pool and demultiplexing the frames is any.



Principle of the cIPS engine

3.1 Integration into your application

The library via netif_dispatch(), needs to be called repeatedly by the Application. In my configuration without operating system, I integrated it into a while loop where I do many other things.

```
while(1)
{
    err = netif_dispatch(netif_adapter);
    if(err) {printf(("%s \r\n", get_last_stack_error( netif_adapter, err)));}
    ....
}
```

Example of integration: connection_manager.c, main.c

3.2 Error reporting

In the flow of demulteplexing and processing, all functions return an error code and the last error if any is parsed by get_last_stack_error(). The possible errors are listed in err.h and are most likely an alteration of the incoming frame (wrong check sum) or a resource limit reached.

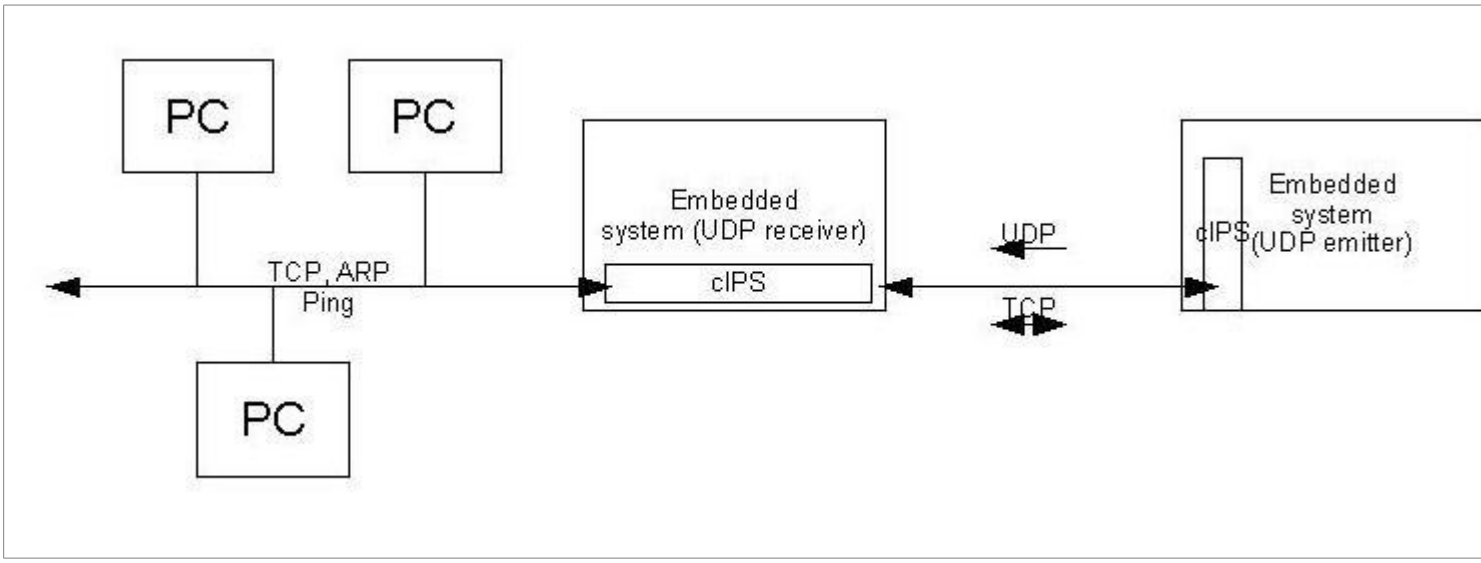
3.3 Several adapters.

Data are private to an adapter. So if you need two adapters, just declare two network adapter instances.

```
  NETIF_T* netif_adapter[2];

  netif_adapter[J1] = na_new(mac_address1, ip_address1, ...
  netif_adapter[J2] = na_new(mac_address2, ip_address2, ...

  while(1)
  {
    err = netif_dispatch(netif_adapter[EU_J2]);
    if(err) {DT_ERROR(("%s \r\n", get_last_stack_error( netif_adapter[EU_J2],
err)));}

    err = netif_dispatch(netif_adapter[EU_J1]);
    if(err) {DT_ERROR(("%s \r\n", get_last_stack_error( netif_adapter[EU_J1],
err)));}

    ...
  }
```

4. Optimization

One of my requirements was to support two network adapters but another important one was to send data every millisecond via UDP between to embedded systems running at 50Mhz and to support TCP on top.

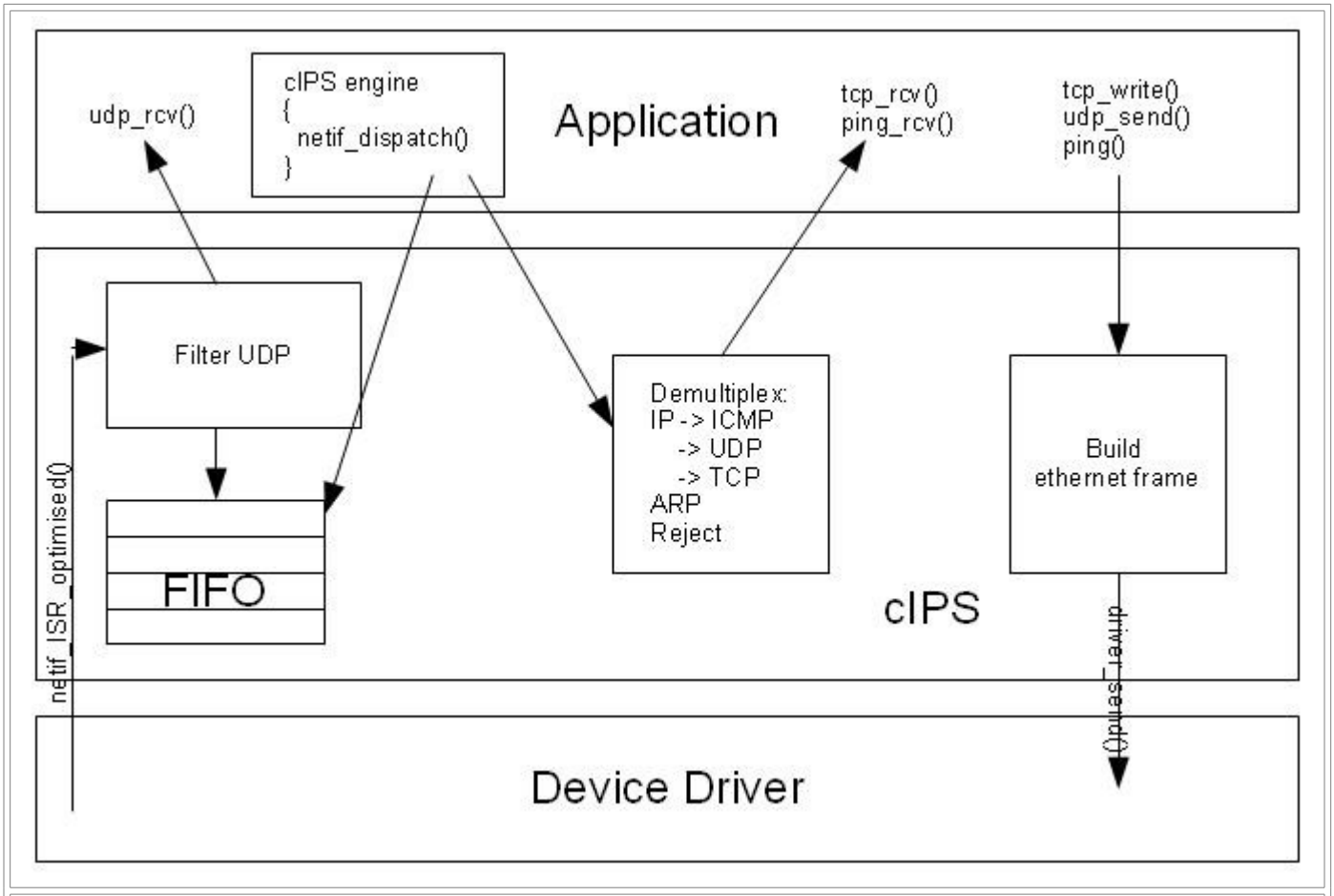


System configuration

4.1 UDP receiver

To process UDP frames as they come and as quickly as possible, UDP frames were parsed and process in the ISR, the other ones (TCP, ARP, ICMP) were put into the FIFO.

To set that mode, set the optimized parameter of na_new() to TRUE. It will replace netif_ISR() is replaced by netif_ISR_optimized().

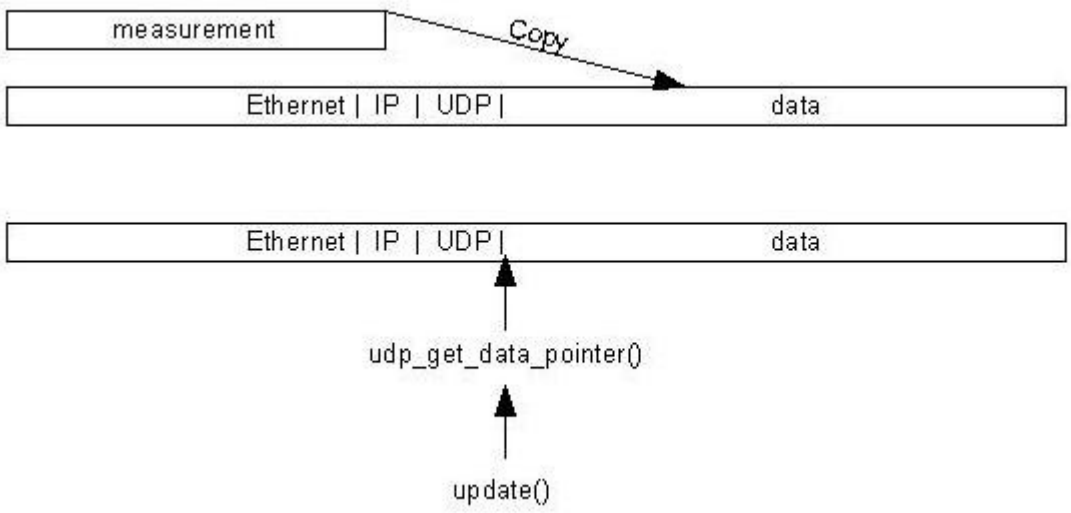


UDP in the ISR

4.2 Sending UDP data

In a traditionnal case, a pointer to some data is passed as a parameter to udp_send() and these data are copied to build an ethernet frame.

```
MEASUREMENT_T measurement;
update(&measurement);
err = udp_send( udp_c, &measurement, sizeof(MEASUREMENT_T),  FALSE);
```



In order to save that copy, it is possible to get a pointer to the data portion of the UDP frame and to work with that reference.

```
MEASUREMENT_T* measurement = (MEASUREMENT_T*)udp_get_data_pointer(udp_cb);
update(measurement);
err = udp_send( udp_c, NULL, sizeof(MEASUREMENT_T),  FALSE)
```

4.3 Sending the same amount of data to the same destination

Sometimes, an UDP connection is used to send repeatedly the same structure to the same destination. It would be inefficient to re-build the ethernet frame from scratch when most of the field do not changed. To address that case, set the "reuse" of udp_send() to TRUE.

```
//err_t udp_send(UDP_T *udp_c, void* data, u32_t data_length, u32_t reuse)
err = udp_send( udp_c, NULL, sizeof(MEASUREMENT_T),  TRUE)
```

4.6 Point to point

Application data are protected by check sums at three levels: Ethernet, IP and UDP.
In a point to point configuration, the UDP check sum is redundant and its generation can be avoided to save some CPU processing.
To inhibit the UDP check sum generation, set the point_to_point parameter of udp_new() to TRUE.

```
  //UDP_T * udp_new( u32_t ipaddr, u16_t port, u32_t point_to_point);
  udp_cb = udp_new(ip_addr, port, TRUE);
```

5. Device driver API requirements

cIPS is compatible with device drivers that respect the following interface:

- u32_t driver_receive(void* pDriver_arg, u8_t *eth_frame);
- u32_t driver_send(void* pDriver_arg, u8_t *eth_frame, u32_t byte_count);
- void _ISR(void *network_adapater);

Otherwise an adaptation layer is needed.

Example of adaptation layer: network_adapter.c, network_adapter.h

Author:
Jean-Marc David
jmdavid1789<at>googlemail.com

cIPS Release Tag:

Name
CIPS_03_000

file version:

Id
cips.h,v 1.5 2010/06/25 14:14:30 jdavid Exp
