Single interface and a single IP-address.

Up until now, FreeRTOS+TCP could handle one Network Interface and a single IPv4 address.

The IP-task is started up with this call:

```
FreeRTOS_IPInit( ucIPAddress, ucNetMask, ucGatewayAddress, ucDNSServerAddress, ucMACAddress );
```

These parameter values will be used as a default. If DHCP is configured, a different set of parameters may be obtained and used.

The network parameters are stored in 3 hidden/global structures:

```
xNetworkAddressing              // Network parameters as they will be used
xDefaultAddressing              // Network parameters default values, as provided to FreeRTOS_IPInit()
xDefaultPartUDPPacketHeader     // Default values for UDP header, contains the IP-address
```

Every Network Interface driver has 3 access functions:

```
BaseType_t xNetworkInterfaceInitialise( void );
BaseType_t xNetworkInterfaceOutput( NetworkBufferDescriptor_t * const pxNetworkBuffer,
    BaseType_t xReleaseAfterSend );
BaseType_t xGetPhyLinkStatus( void );
```

Only `xGetPhyLinkStatus()` may be used by an application, the other functions are private to the library.

The `xNetworkInterfaceInitialise()` function is called from `prvProcessNetworkDownEvent()`. As long as it returns `pdFAIL`, a new network-down event will be scheduled after roughly 3 seconds. This delay can be configured by changing `ipINITIALISATION_RETRY_DELAY` (unit : clock ticks).

Here is a typical example:

```
BaseType_t xNetworkInterfaceInitialise()
{
static BaseType_t xHasInit = pdFALSE;
static BaseType_t xInitResult = 0;
BaseType xResult = pdFAIL;

    if( xHasInit == pdFALSE )
    {
        xHasInit = pdTRUE;
        xInitResult = emac_init();
    }
    if( xInitResult == pdPASS )
    {
        if( check_phy_status() != 0 )
        {
            xResult = pdPASS;
        }
    }
    return xResult;
}
```

If DHCP is to be used, the interface will apply for an IP-address, and get the necessary network information like the net-mask, a DNS server, and a gateway.

If DHCP is not used, or after it is ready, this internal function is called:

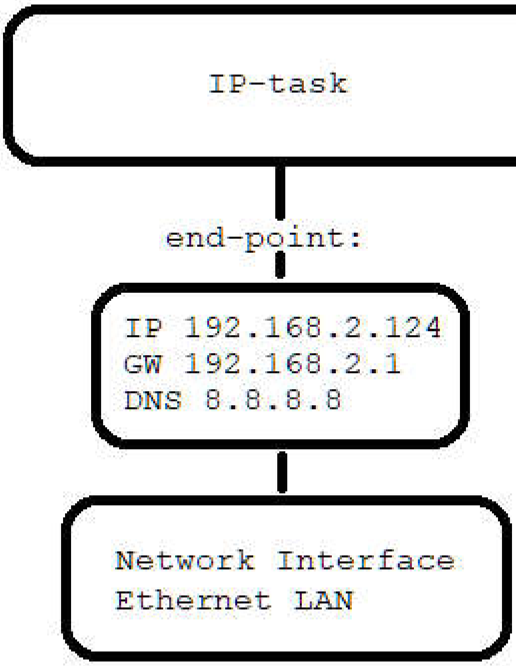
```
void vIPNetworkUpCalls( void );
```

which will call the application hook:

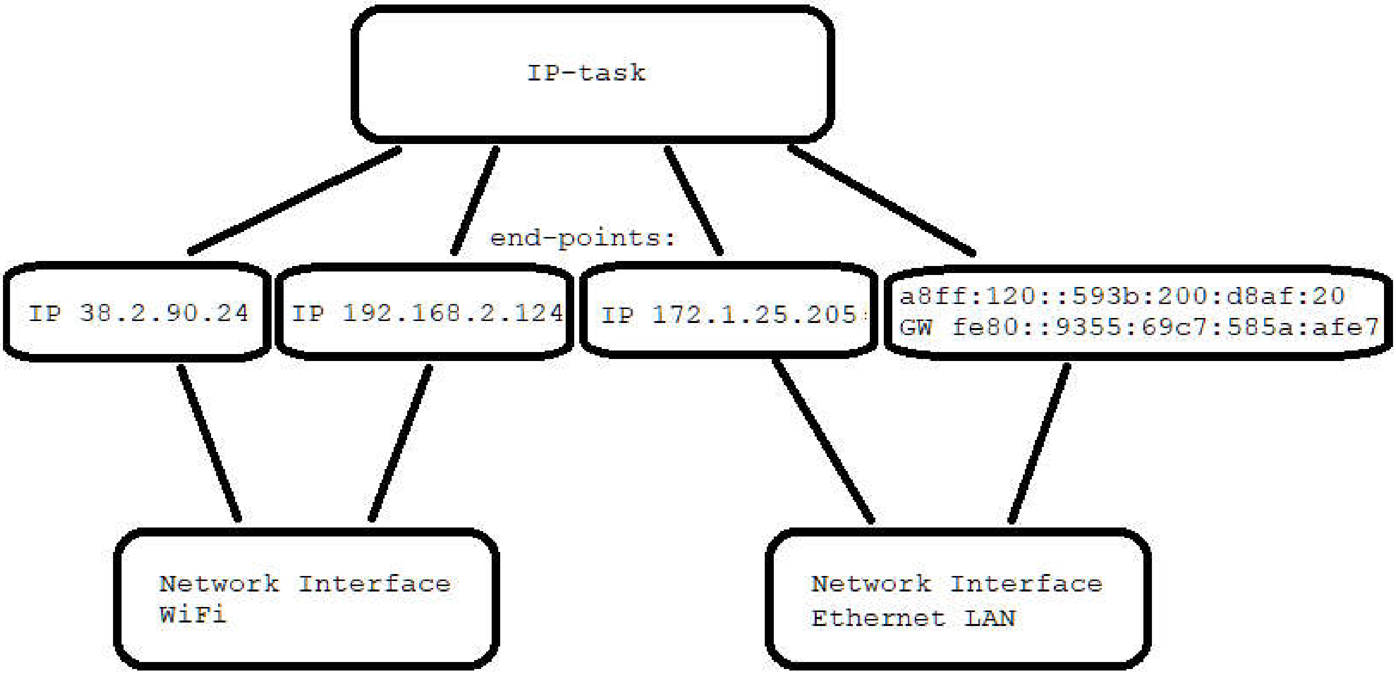
```
void vApplicationIPNetworkEventHook( eIPCallbackEvent_t eNetworkEvent );
```

The event type will be '`eNetworkUp`' to tell the application that the IP-task is ready for communication. From then on, the application can create sockets and start communication.

The earlier single interface / single end-point library can be represented like this:



The new library with multiple interfaces and multiple end-points is structured like this:



Multiple interfaces and multiple end-points.

The earlier call `FreeRTOS_IPInit( IP, mask, gw, dns, mac )` will be replaced by:

```
BaseType_t FreeRTOS_IPStart( void );
```

Before calling this function, the application must add Network-Interfaces and so-called End-Points.

A Network-Interface is the driver of the EMAC + PHY (or a WiFi adapter). An end-point is set of network parameters: an IP-address, MAC-address, Gateway, DNS, etc. There can be multiple Interfaces, each one owning one or more end-points.
DHCP : when enabled, each end-point will have its own DHCP session. Both DHCPv4 and DHCPv6 are available.

Here is a complete example that initialises two Zynq interfaces, where each interface has one end-point:

```
/* Declare 2 network interface descriptors: */
static NetworkInterface_t xInterfaces[ 2 ];
/* Declare 2 end-points: */
static NetworkEndPoint_t xEndPoints[ 2 ];

/*-----------------------------------------------------------*/

/* Bind the address functions for EMAC-0. */
pxZynq_FillInterfaceDescriptor( 0, &( xInterfaces[ 0 ] ) );
/* Fill the IPv4 end-point structure. */
FreeRTOS_FillEndPoint( &( xEndPoints[ 0 ] ), ucIPAddress, ucNetMask,
    ucGatewayAddress, ucDNSServerAddress, ucMACAddress );

/* You can modify fields, enable DHCP: */
xEndPoints[ 0 ].bits.bWantDHCP = pdTRUE;

/*-----------------------------------------------------------*/

/* Bind the address functions for EMAC-1. */
pxZynq_FillInterfaceDescriptor( 1, &( xInterfaces[ 1 ] ) );
/* Fill the IPv6 end-point structure. */
FreeRTOS_FillEndPoint_IPv6( &( xInterfaces[0] ),
                            &( xEndPoints[ 1 ] ),
                            &( xIPAddress ),      /* The default IP-address to use. */
                            &( xPrefix ),          /* Network prefix to be used. */
                            64uL,                  /* Prefix length. */
                            &( xGateWay ),         /* Gateway to the web. */
                            &( xDNSServer ),
                            ucMACAddress1 );       /* Bind it to this MAC-address. */
/* Let it use Router Advertisement ( SLAAC ) */
xEndPoints[ 1 ].bits.bWantRA = pdTRUE;
/* Start the IP-task. */
FreeRTOS_IPStart();

/*-----------------------------------------------------------*/
```

The structures (`xInterfaces` and `xEndPoints`) must **remain to exist** as long as the IP-stack runs. So it is not a good idea to put them on the stack of `main()`, because the stack may get reused for other purposes (ISR stack), once the scheduler is running.

Here is a summary of the starting-up of multiple interfaces and end-points:

- `pfInitialise()` is being called repeatedly for every interface until it returns `pdPASS`.
  An interface is said to be "up" when it is initialised, and when its Link Status is high.

- Once it is up, for each end-point that is bound to that interface:
  `vDHCPProcess()` / `vDHCPv6Process()` (an internal function) will be called repeatedly for every end-point that has DHCP enabled (IPV4/IPV6)
  `vRAProcess()` (an internal function) will be called repeatedly for every end-point that has SLAAC enabled (IPV6)

- As the last step, a network-up event is generated for every end-point:
  `vApplicationIPNetworkEventHook( eNetworkUp, pxEndPoint );`
  In FreeRTOS+TCP, the entire IP-stack was said to be UP or DOWN. This could be tested by calling the function `FreeRTOS_IsNetworkUp()`.

In the /multi version, an **end-point** is UP or DOWN. This can be checked with:

```
/* Return true if a given end-point is up and running.
When FreeRTOS_IsEndPointUp() is called with NULL as a parameter,
it will return pdTRUE when all end-points are up. */
BaseType_t FreeRTOS_IsEndPointUp( struct xNetworkEndPoint *pxEndPoint );
```

Check if all end-points belonging to a given interface are up:

```
/* Return pdTRUE if all end-points are up.
When pxInterface is null, all end-points can be iterated. */
BaseType_t FreeRTOS_AllEndPointsUp( NetworkInterface_t *pxInterface );
```

Changes to NetworkInterface.c :

The three access functions of a Network Interface are declared **`static`** and they get `pxInterface` as the first parameter:

```
static BaseType_t xZynqNetworkInterfaceInitialise( NetworkInterface_t *pxInterface );

static BaseType_t xZynqNetworkInterfaceOutput( NetworkInterface_t *pxInterface,
                                               NetworkBufferDescriptor_t * const pxBuffer,
                                               BaseType_t bReleaseAfterSend );

static BaseType_t xZynqGetPhyLinkStatus( NetworkInterface_t *pxInterface );
```

The prefix "**`xZynq`**" is just an example, taken from the Xilinx Zynq Network Interface.

The addresses of these 3 functions will be bound to the Interface descriptor in the function **`pxZynq_FillInterfaceDescriptor()`**:

```
    pxInterface->pfInitialise         = xZynqNetworkInterfaceInitialise;
    pxInterface->pfOutput             = xZynqNetworkInterfaceOutput;
    pxInterface->pfGetPhyLinkStatus   = xZynqGetPhyLinkStatus;
```

Note that the functions **`pfInitialise`** and **`pfOutput`** are not public, they will be called by the IP-task only.

Routing implementation

Incoming packets:

For every incoming packet, two fields will be set in the Network Buffer:

```
pxNetworkBuffer->pxInterface = pxMyInterface;
pxNetworkBuffer->pxEndPoint = FreeRTOS_MatchingEndpoint( pxMyInterface,
   pxBufferDescriptor->pucEthernetBuffer );
```

The field `pxInterface` tells on which interface the packet has arrived. This is important when replying to the message.
The reason for setting `pxInterface` is of course to remember on which network interface the packet was received. In most cases it will be returned on the same interface.

The field `pxEndPoint` tells which return IP-address shall be used when replying.

The function `FreeRTOS_MatchingEndpoint` will try to find the best matching end-point, by looking at both the destination and source addresses:

- ARP packets (IPV4 only):

If it is an ARP packet: take an end-point whose IP-address is equal to the **ARP target address**.
Other ARP packets can be ignored.

- Neighbour Solicitation (IPv6 only)

Find an end-point with an IP-address that is the same as the target address in the ICMP packet.

- Unicast messages:

Find a matching end-point with the same IP-address as the **target address** in the IP-header

- Broadcast messages (IPV4 only):

Find a matching end-point with an IP-address that is in the same network as the **target address** in the IP-header,
E.g. 192.168.2.**255** matches with 192.168.2.**10**
255.255.255.255 matches with any IP-address, and the first end-point will be taken.

When no matching end-point is found, NULL will be returned, and the packet may be dropped.

The field **`pxNetworkBuffer->pxEndPoint`** is set in order to remember which source IP-address shall be used when a reply is sent.

IP-task reception:

A `eNetworkRxEvent` will pass the received packet to the IP-task.

prvProcessEthernetPacket send the packet either to:

eARPProcessPacket();

A reply will be stored into the ARP cache table, along with the end-point set by the driver.
A request, if matching, will be replied to, using the end-point set by the driver.
vReturnEthernetFrame()
Assumes the NetworkBuffer->pxEndPoint has been set

prvProcessIPPacket();

TCP :

vReturnEthernetFrame() is called from:
prvReplyDNSMessage()
prvProcessEthernetPacket() return an ARP, UDP, or TCP packet
prvReturnICMP_IPv6()

IP-task sending a UDP message

The user calls FreeRTOS_sendto() with a sockaddr
A `eStackTxEvent` will pass the packet-to-be-sent to the IP-task.

vProcessGeneratedUDPPacket()
Multicast:
If the end-point is known, use the broadcast MAC-address.
If not, see Unicast here below
Unicast:
Look-up the IP-address in the ARP table. This also results in an end-point

If the lookup failed:
turn the packet into an ARP packet and send it.
Use `FreeRTOS_FindEndPointOnNetMask()` to find the correct end-point

Call `pxInterface->pfOutput()` directly to send the packet.