Adding a Second Ethernet Driver to NET+OS

Table of Contents

Adding a Second Ethernet Driver to NET+OS

Overview

This application note describes how to:

- Interface a second Ethernet driver with the Fusion stack.
- Make the driver compatible with the memory management functions supported on ARM9-based processors.
- Update the NET+OS startup code to add the new Ethernet interface to the network device table.
- Update ACE to automatically configure a second network interface.

Limitations

The Fusion stack requires the two Ethernet interfaces to be on separate subnets.

Writing the Ethernet driver

Fusion Stack Interfaces

To interface with the Fusion TCP/IP stack, the driver must provide functions that the stack can call to:

- Initialize the driver.
- Enable and disable the driver.
- Send a packet.
- Perform I/O control operations.

The driver must call functions in Fusion to transfer receive packets to the stack and to allocate and free packet buffers.

Initialization Function

Fusion calls the initialization function once. The function prototype is:

```
int initFunction (struct netdev * ndp);
```

The *netdev* structure is defined in the `netdev.h` header file in the `h/tcpip` directory.

The function must:

- Save the value of the *ndp* argument. This value is a pointer to the device's entry in the network device table. The Ethernet driver uses this value to process other calls from the TCP/IP stack, and to call functions in the TCP/IP stack.
- Initialize the Ethernet hardware.
- Initialize the rest of the Ethernet driver, including:
  - Creating any threads or other OS resources the driver will need
  - Allocating any buffers the driver will need
- Copy the interface's Ethernet MAC address into the *ndp->nd_11addr.a_ena* array.
- Set *ndp->nd_11addr.a_type* to `AF_ETHER`.
- Set *ndp->nd_11addr.a_len* to `6`.
- Set the transmit mode the driver will use.
- Return either of these values:
  - If the driver is initialized: `0`
  - If the driver isn't initialized: `-1`

The initialization function must set the transmit mode by calling one of these functions:

- `FUSION_DRV_BLOCK_UNTIL_TXCOMPLETE(`*ndp*`)`: The driver's transmit function does not return until after the packet is completely transmitted. The stack can free or reuse the packet buffer when the transmit function returns and can call the transmit function again with another packet to transmit.
- `FUSION_DRV_TXCOMPLETE_NOTIFY_COPY(`*ndp*`, `*limit*`)`: The transmit function copies the transmit packet to an internal buffer and places it in an internal transmission queue. The function returns before the packet is completely transmitted. The stack frees or reuses the packet buffer when the transmit function returns. The Ethernet driver notifies the stack when the transmit is complete by calling `ndq_restart(`*ndp*`)`. The stack continues to call the Ethernet driver's transmit function until either the driver's queue is full or the stack has no more packets to transmit.

  The initialization function must set the *limit* argument, which is an int, to the maximum number of transmit packets that can be placed in the driver's transmit queue.

  Because of the performance cost in copying the packet, NetSilicon does not recommend using this mode unless it is required by the Ethernet hardware.
- `FUSION_DRV_TXCOMPLETE_NOTIFY_NOCOPY (`*ndp*`, `*limit*`)`: The transmit function places the packet in an internal transmission queue. The function returns before the transmit operation completes. The Ethernet driver notifies the stack when the transmit is complete by calling `ndq_restart(`*ndp*`)`; the stack can free or reuse the packet buffer at this point. The stack continues to call the transmit function until either the driver's queue is full or it has no more packets to transmit.
- The initialization function must set the *limit* argument to the maximum number of transmit packets that can be placed in the queue. The stack uses the `ndq_restart()` function to keep track of the number of packets in the Ethernet driver's transmit queue and when it can free the current transmit packet. The Ethernet driver must transmit the packets in the order in which the stack provides them.
- `FUSION_DRV_ZEROCOPY (`*ndp*`, `*limit*`)`: This mode operates in the same way as `FUSION_DRV_TXCOMPLETE_NOTIFY_NOCOPY`, with this difference: the Ethernet driver can transmit packets that are broken into more than one buffer.
- `FUSION_DRV_NEAR_ZEROCOPY (`*ndp*`, `*limit*`)`: This mode operates in the same way as `FUSION_DRV_ZEROCOPY`, with this difference: the stack ensures that all buffers are aligned on 32-bit boundaries.

For an example of how to write an Ethernet driver initialization function, see the `eth0_init()` function `in eth_stack.c` and the `eth_init()` function in `eth_init.c`.

Transmit Function

The stack calls the Ethernet driver's transmit function to send a packet. The function prototype is:

```
int transmitFunction(struct m * mp);
```

The *m* structure and macros to manipulate it are defined in the `m.h` file in the `h/tcpip` directory. For additional information, see the online help.

This function needs to transmit the packet (or initiate its transmission). The stack calls this function with a pointer (*mp*) to a message (*m*) structure. A reference to the appropriate device table entry is defined in the *m_ndp* member of the message structure. (If the driver supports one Ethernet interface, *mp->m_ndp* is set to the value of the *ndp* argument passed to the initialization function).

The size of the packet can be determined with the `m_dsize(`*mp*`)` macro. If the packet is smaller than the minimum packet for the network, you want the driver (or the Ethernet hardware) to pad the packet to the minimum length. The *m_hp* member of the message structure points to the beginning of the packet. The function returns either `0` on success or `-1` on failure.

- If the `FUSION_DRV_BLOCK_UNTIL_TXCOMPLETE`, `FUSION_DRV_TXCOMPLETE_NOTIFY_COPY`, or `FUSION_DRV_TXCOMPLETE_NOTIFY_NOCOPY` transmit mode was specified by the initialization function, the packet is stored in a single buffer that `mp->m_hp` points to. The macro call `m_dsize(`*mp*`)` returns the length of the packet.
- If `FUSION_DRV_ZEROCOPY or FUSION_DRV_NEAR_ZEROCOPY` mode was specified, the packet can be in more than one buffer. The *mp->m_hp* field stores a pointer to the first buffer, whose length can be determined by subtracting *mp->m_hp* from *mp->m_ptail*.

  The *mp->m_cell_count* field indicates the number of additional buffers (if any) in the packet. The additional buffers are stored in the *mp->m_datacells* array. A pointer to each of the additional packet buffers is stored `in` *mp->m_datacells[index].celldata*, with the length stored in *mp->m_datacells[index].cellsize*, where *index* is a zero-based array index.
- If the `FUSION_DRV_BLOCK_UNTIL_TXCOMPLETE` transmission mode was specified by the `init` function, this function must initiate transmission of the packet and wait until the packet has been fully transmitted before returning to Fusion. On return, Fusion assumes that the packet has been transmitted and frees its memory. Fusion doesn't need to receive a transmit complete notification.

- If one of the other transmission modes was specified by the initialization function, the transmit function should return after starting transmission of (or queuing) the packet. The stack can submit multiple packets to the driver (each by a separate call this function), up to the limit specified in the macro by which the mode was selected. For each packet that completes transmission, the driver must notify the stack by calling the `ndq_restart()` function. Fusion applies this notification to packets in the order in which packets were submitted to the driver. The `ndq_restart()` function must be called from a thread.
- If the `FUSION_DRV_TXCOMPLETE_NOTIFY_NOCOPY`, `FUSION_DRV_ZEROCOPY`, or `FUSION_DRV_NEAR_ZEROCOPY` mode was specified, Fusion preserves the packet in memory until a transmit complete notification has been received for that packet. From a performance standpoint, this is the preferred mode of operation.

Because Fusion calls the transmit function from within a critical section, keep the transmit function as short as possible and do not use ThreadX functions. Complex operations should be handed over to another thread. Fusion assumes that this function always succeeds, and it does not check the return value.

For an example of an Ethernet driver transmit function, see the `eth0_start()` function in `eth_stack.c`. For an example of a routine that notifies Fusion when a packet has been completely transmitted, see the `eth_restart()` function in `eth_stack`.

Ioctl function

The stack calls this function to perform I/O control operations. The prototype for this function is:

```
int ioctlFunction(struct netdev * ndp, int cmnd, char * addr);
```

Fusion calls this function with a pointer (*ndp*) to the device table entry for the device. The *cmd* argument is set to a device-specific I/O control. The codes shown next are defined for Ethernet I/O control operations. These codes select an action in the I/O control routine.

Code	Action selected
`ENIOCNORMAL`	Receive broadcast and unicast packets.
`ENIOCPROMISC`	Receive all undamaged packets with any destination address.
`ENIOCALL`	Receive all packets.
`ENIOCRESET`	Reset the device.
`ENIOCWHATRU`	Return the device name.
`ENIOADDMULTI`	Add a multicast address.
`ENIODELMULTI`	Delete a multicast address.

Use the *addr* argument only in conjunction with the IGMP protocol to add and delete multicast addresses from a device. In this case, the argument points to a 6-byte buffer that contains the Ethernet multicast address to add or delete.

For an example of an Ethernet I/O control function, see `eth0_ioctl()` in `eth_stack.c`.

Enable and Disable function

The stack calls this function to enable or disable the interface. The function prototype is:

```
int enableDisableFn(struct netdev *ndp, u16 flags, char
*options);
```

Fusion calls this function with a pointer (*ndp*) to the device table entry for the device. Set the *flags* argument to `1` (*flags*=1) to bring a device up or `0` (*flags*=0) to bring the device down.

The *options* argument is reserved for future use; ignore this argument.

This function should return either `0` to indicate success or `-1` to indicate failure. When this function is called to bring a device on-line, it activates the device to start sending and receiving data.

Fusion calls this function from within a critical section. The function must not use ThreadX function calls or block.

For an example of this function, see the `eth0_updown()` function.

Receive operation

The Ethernet hardware DMAs the receive packets into memory. The Ethernet driver must either:

- Set up the hardware to DMA the packets directly into buffers allocated using the stack's `m_new()` routine. (See the next section, "Allocating Packet Buffers.")
- Copy packets into these buffers before passing them up to the Fusion stack.

  After a packet has been received, the driver must examine the packet to determine whether it should be passed up to the TCP/IP stack. Packets should be passed up only if they meet both of these conditions:

  – They are encapsulated in Ethernet-II frames.

  – The packet type field is set to one of the IP packet types. The IP packet types are `0x800` (IP), `0x806` (ARP), and `0x8035` (RARP). Other packets should be either discarded with the `smkill()` function (see the section "Freeing Packet Buffers") or passed to other protocol stacks.

To indicate the beginning and end of the packet data, the driver must adjust the head and tail pointers of the *m* structure. Assuming that the pointer to the *m* structure is named *mp*, the head pointer, *mp->m_hp*, must point to the beginning of the packet, and the tail pointer, *mp->m_tp*, must point to the end of the packet. The field *mp->m_ndp* must be set to the value of the *ndp* argument that was passed to the initialization function.

The Ethernet driver uses the `msm()` function to pass packets to the stack. The function prototype is:

```
void msm(struct m *mp, en_up);
```

A pointer to the packet's m structure should be passed in the first argument, and the second argument should always be set to `en_up`, which is defined in `h/tcpip/enet.h`. The Ethernet driver must not use the packet buffer after calling this function. The stack frees the buffer when it finishes processing it.

The driver should pass packets up to the stack only if the driver is enabled. Any packets received while the driver is disabled should be discarded.

The `msm()` routine must be called from a thread, not from an ISR.

For a sample implementation of the Ethernet receive operation, see the code in `eth_recv.c` and the `eth_recv_up()` function in `eth_stack.c`. Be aware that this code is specific to the NET+ARM Ethernet hardware module.

Memory allocation

Allocating Packet Buffers

The driver must use the `m_new()` function to allocate packets buffers. The prototype for this function is:

```
struct m *m_new(int size, NULL, F_NOBS);
```

The *size* argument indicates the specified buffer size in bytes. The second and third arguments must be set to `NULL` and `F_NOBS`, respectively. `F_NOBS` is defined in `h/tcpip/flags.h`. If `m_new` succeeds, it returns a pointer to an `m` structure, which contains the buffer. If `m_new` is unable to allocate the buffer, it returns `NULL`.

When a buffer is allocated with `m_new()`, both the head and tail pointers point to the end of the packet buffer. The Ethernet driver must adjust the head pointer so it points to the beginning of the buffer. Assuming that `mp` is an m structure pointer, and `packetLength` is the specified length of the packet, the code to allocate a packet is:

```
mp = m_new(packetLength, NULL, F_NOBS);
if (mp == NULL)
{
     /* code to handle out of memory error goes here*/
}
else
     {mp->m_hp -= packetLength;
}
```

For an example of a packet allocation routine, see the `NAEthMsgAllocate()` routine in `eth_stack.c`.

Freeing Packet Buffers

The Ethernet driver should use the `smkill()` function to free buffers allocated with `m_new()`. Note that received packets passed up to the stack and transmit packets are automatically freed by the stack and should not be freed by the Ethernet driver.

The prototype for `smkill()` is:

```
void smkill(struct m *mp);
```

The *mp* argument should be set to the `m` structure pointer returned by `m_new()`.

For an example of a routine that frees a packet buffer, see the `NAEthMsgFree()` function in `eth_stack.c`.

MMU Concerns

Netsilicon's ARM9-based processors have a Memory Management Unit (MMU). Support for the MMU was added in NET+OS 6.1. If you are using NET+OS 6.1 or later, you must write your Ethernet driver so it can work with the ARM9 MMU.

The MMU Translation Table

The Control and Status Registers (CSRs) for the second Ethernet interface appear somewhere in the system address space. The MMU uses an address translation table to determine which areas of the address space can be accessed by the CPU. You need to add the section of memory used by the second Ethernet interface's CSRs to this table.

If the Ethernet interface is connected through PCI, and the NET+OS PCI API is used to set it up, the PCI API automatically updates the MMU's translation table as necessary. In such a case, you can skip the rest of this section. Otherwise, you must update the translation table yourself if the Ethernet interface is connected to the processor through some other mechanism (a chip select, for example).

The `customizeCache.c` file in the BSP platform directory contains an array, `mmuTable` that is used to build the MMU address translation table. The array entries include:

- The start and stop addresses of the memory region.
- The page size of the memory region, which determines the granularity of the region's address map. 1 MB, 64K, and 4K page sizes are supported.
- The cache mode for the memory region, which determines the type of buffering and caching done for the memory region.
- The access mode, which determines the types of accesses (reads and writes) that are allowed to the memory region.

Memory regions that are not listed in this array are assigned the `MMU_NO_ACCESS` access mode. A data abort exception is generated if the processor tries to access memory assigned this access mode. The default entries in this table set up the memory map for the processor's CSRs, as well as ROM and RAM.

You must add an entry to the `mmuTable` array for the section of memory occupied by the second Ethernet interface's CSRs; otherwise, the MMU generates data abort exceptions when your driver accesses it. Set up CSRs with the cache mode set to `MMU_BUFFERED` and the access mode set to `MMU_CLIENT_RW`. The page size is usually set to `MMU_PAGE_SIZE_4K` if the memory region contains only CSRs. However, you may want to set the page size to `MMU_PAGE_SIZE_64K` if the module has more than 64K of CSRs (a piece of shared RAM, for example).

For example, suppose the CSRs for the second Ethernet interface are mapped to addresses `0xB0000000` to `0xB00001F0`. You would add this entry to `mmuTable`:

```
{0xB0000000, 0xB00001F0, MMU_PAGE_SIZE_4K, MMU_BUFFERED,
MMU_CLIENT_RW}
```

The Ethernet hardware should signal a data abort if code with errors attempts to access a nonexistent CSR; otherwise, the processor (and JTAG debugger) can hang. If the second Ethernet module does not signal a data abort, you can use the MMU to prevent such a hang by using the smallest page size (`MMU_PAGE_SIZE_4K`) for CSRs. Using this page size breaks up that portion of the address space into 4K chunks. The only accesses that can hang the processor are those to addresses that are between the actual end of the CSRs and the end of the page that contains them. This is because these are the only addresses with the access mode of `MMU_CLIENT_RW` that map to nonexistent memory.

For detailed information about `mmuTable` array and the MMU API, see the online help.

DMA Buffer Descriptors

DMA buffer descriptors *must not* be in cached memory. You must use the `nonCachedMalloc()` function to allocate memory used for DMA buffer descriptors; failing to do so causes intermittent problems that are difficult to debug. The `nonCachedMalloc()` and `nonCachedFree()` functions are described in the online help.

For an example of how to use `nonCachedMalloc()`, see the `eth_allocate_rx_buffer_descriptors()` function in `eth_dma.c`.

DMA To or From a Cached Buffer

The buffers that store Ethernet packets can be in cached memory. Cached buffers must be cleaned and/or invalidated before data is DMAed to or from them, ensuring that the contents of cache are in sync with the contents of the buffer. Use the `NABeforeDMA()` and `NAAfterDMA()` functions for this purpose. Use the `NABeforeDMA()` function to clean and invalidate buffers before a DMA operation, and call the `NAAfterDMA()` function after a DMA operation completes. These functions are described in detail in the online help.

For example, this would be used to prepare a transmit buffer for DMA and process the buffer after the DMA engine finishes reading it:

```
NABeforeDMA(NA_MEMORY_TO_DEVICE, buffer, NULL, length);
/** code to DMA the transmit buffer goes here **/
/** the call below should be executed after DMA completes **/
NAAfterDMA(NA_MEMORY_TO_DEVICE, buffer, NULL, length);
```

DMA and Virtual Addresses

The MMU can remap memory to different locations in RAM. For example, the MMU can map a piece of RAM located at physical address X to virtual address Y. The pointers to packet buffers given to the Ethernet driver are virtual addresses, not physical addresses. However, the Ethernet driver must supply the DMA engine with the physical address of buffers, not their virtual address. The Ethernet driver must use the `NAVaToPhy()` function to convert the packet buffer pointers passed to it, which are virtual addresses, into physical addresses when it constructs the DMA buffer descriptors.

For example, suppose *`mp`* is a pointer to an *`m`* structure with a single transmit packet buffer, and *`dmaDesc`* is a pointer to an *`fb_buffer_desc_t`* DMA descriptor structure.

The code to set up dmaDesc would be:

```
void *virtualAddress = mp->m_hp;
physical_address_t physicalAddress;
int length = m_dsize(mp);
volatile NA_DMA_DESCRIPTOR_TYPE *dmaDesc;
dmaDesc = nonCacheMalloc(sizeof(fb_buffer_desc_t));
if (NAVaToPhys(virtualAddress, &physicalAddress) == FALSE)
{
     /** code to handle error goes here **/
}
NABeforeDMA(NA_MEMORY_TO_DEVICE, virtualAddress, NULL, length);
dmaDesc->src_addr = physicalAddress;
dmaDesc->dst_addr = 0;
dmaDesc->buf_len = length;
```

For detailed documentation about the NAVaToPhy() function, see the online help.

Integrating into the NET+OS Startup Code

You must update the NET+OS startup code to add the Ethernet device into Fusion's table of network devices. After you add the device to the table, the device can be accessed as a Fusion network interface. You also need to modify the startup code to acquire and configure a network configuration for the interface.

Create a Device Driver Table Entry

The `netosStartTCP()` function in `starttcp.c` is called during system startup to bring up the Fusion stack and the network devices. Modify this function to add your Ethernet driver into Fusion's device table.

The first step is to create a *`NADeviceTableEntryType`* structure with information about the device. The structure is defined as:

```
typedef struct {
   char devName[MAX_DEV_NAME];
   u16 devDevid;
   u16 devXflags;
   u32 devParam0;
   u32 devParam1;
   u32 devParam2;
   u32 devParam3;
   u16 devFamily;
   NADevInitFnType devInit;
   NADevUpDownFnType devUpdown;
   NADevAddLinkLayerFnType devAddLinkLayer;
   NADevTransmitFnType devTransmit;
   NADevIoctlFnType devIoctl;}
NADeviceTableEntryType;
```

Set the fields as described in this table:

Field	Setting
devName	Set to the name of the device
devDevid	One of these: - 0, if the Ethernet driver supports a single Ethernet port. - If the driver supports more than one port, create a *NADeviceTableEntryType* structure for each port, and set this field to a zero-based index to each port.
devXflags	One of these: - F_X_MULTI_ENBL if the Ethernet driver supports the ENIOADDMULTI and ENIODELMULTI I/O control commands - 0 if the driver does not support multicast.

devParam0 – devParam3	The *devParam3* fields hold context information for the device driver. If you do not need these fields, you can leave them uninitialized.
devFamily	AF_ENET.
devInit	The address of the Ethernet driver's initialization function.
devUpdown	The address of the Ethernet driver's enable/disable function.
devAddLinkLayer	The address of a routine that will construct the link layer packet headers. For Ethernet devices, set this field to en_scomm, which is defined in h/tcpip/enet.h.
devTransmit	The address of the Ethernet driver's transmit.
devIoctl	The address of the Ethernet driver's I/O control function.

This example shows what you want your table entry to look like:

```
static NADeviceTableEntryType eth1 =
{
     "eth1",            /* name */
     0,                 /* device minor ID */
     F_X_MULTI_ENBL,    /* driver supports multicast */
     0,                 /* OEM parameter 0 (not used) */
     0,                 /* OEM parameter 1 (not used) */
     0,                 /* OEM parameter 2 (not used) */
     0,                 /* OEM parameter 3 (not used) */
     AF_ENET,           /* family */
     eth1_init,         /* initialization function */
     eth1_updown,       /* enable/disable function */
     en_scomm           /* Ethernet link layer headers function
     */
     eth1_start,        /* transmit function */
     eth1_ioctl         /* no ioctl function */
   };
```

For details about the NADeviceTableEntryType structure, see the online help.

Add the Device into Network Device Table

You must insert code into netosStartTCP() to add the device to Fusion's device table. The code must call the NAAddDeviceToTcpipDeviceTable() function. Add this code after netosStartTCP() calls so_initialize(), and before the stack's timer thread is created.

The prototype for NAAddDeviceToTcpipDeviceTable is:

```
BOOLEAN NAAddDeviceToTcpipDeviceTable(NADeviceTableEntryType *entry);
```

where *entry* is a pointer to the device's NADeviceTableEntryType structure discussed in the previous section. The function returns TRUE if it succeeds or FALSE if it fails (that is, if the table is full).

Here's an example of how you want your code to look when you add a device into the network device table:

```
/** The code below should be inserted after the call **/
/** to so_initialize() completes, but before the **/
/** Fusion timer thread is created. **/

if (NAAddDeviceToTcpipDeviceTable(&eth1) == FALSE)
{
       /** code to handle error goes here **/
}
```

The netosStartTCP() function already has code in it to add two PPP devices. Use that code as a template.

For detailed information about the NAAddDeviceToTcpipDeviceTable function, see the online help.

Integrate with ACE

The next step is to integrate your Ethernet driver with the Automatic Configuration Executive (ACE). ACE is responsible for getting a configuration (IP address, subnet mask, and so on.) for each network interface and configuring Fusion with these settings. ACE determines what to do by reading an array of aceConfigInterfaceInfo from NVRAM. The data structure has information that tells ACE which protocols to use to get the IP configuration and the options to use with those protocols. You must update this structure so that ACE configures your network interface as well as the primary one.

CONFIG_ACE_MAX_INTERFACES

This manifest constant, which is defined in `h/tcpip/ace_params.h`, determines how much space is reserved in NVRAM for ACE configuration settings. By default, this constant is set to `2`; if you need more than two interfaces, you must increase this value.

Be aware that a virtual interface is created when the Auto-IP protocol is used. So, if Auto-IP is used with a second Ethernet port, you must set `CONFIG_ACE_MAX_INTERFACES` to at least `3`.

Update the Default ACE Configuration.

ACE uses default configuration settings when NVRAM is uninitialized. These settings are stored in the *`NADefaultEthInterfaceConfig`* array, which is an array of *`aceConfigInterfaceInfo`* structures defined in the `aceParams.c` file in the `platform` directory. This array has one element for each network interface. You must create a new entry in this array that has the default settings for the second Ethernet port.

The first entry in *`NADefaultEthInterfaceConfig`* contains the default settings for the primary Ethernet port. To create an entry for the second Ethernet port, paste a copy of the first entry between the first and second entries. This makes the new entry, which is for the second Ethernet device, the second entry in the array. Change the field `ifname` of this entry to match the second Ethernet interface you set in the interface's *`NADeviceTableEntryType`* structure you created (`eth1`). Update the other fields in the new array element with the default configuration settings for the port. The *`aceConfigInterfaceInfo`* structure is described in detail in the online help. You can probably use the same settings as the primary Ethernet port.

You also must increment the `DEFAULT_NUMBER_OF_INTERFACES` and `AUTO_IP_INTERFACE` manifest constants, which are defined in the `aceParams.c` file as well.

Auto-IP

Auto-IP is a protocol for acquiring an IP configuration even if no servers are available on the network. The Fusion stack allows Auto-IP to be run only on a single interface. You cannot use Auto-IP on two Ethernet interfaces simultaneously.

The last entry in *`NADefaultEthInterfaceConfig`* contains the configuration settings for Auto-IP. Set the field `ifname` in this entry to the name of the network interface Auto-IP runs on concatenated with `:0`. For example, `ifname` must be set to `eth0:0` to specify the primary Ethernet interface, which is named `eth0`. If you want to use Auto-IP on the second Ethernet port and you named that port `eth1`, change the `ifname` field to `eth1:0`.

customizeAceGetDefaultEthernetConfig ()

Each application has a configuration file named `appconf.h`. This file defines the `APP_IP_ADDRESS`, `APP_IP_SUBNET_MASK`, and `APP_IP_GATEWAY` manifest constants that set the default IP address, subnet mask, and gateway for the primary Ethernet interface. The `APP_USE_STATIC_IP` manifest constant determines whether the values defined by the other constants should be used instead of trying to get an IP address from a network server.

The `customizeAceGetDefaultEthernetConfig()` and `customizeAceGetDefaultConfig()` functions in `aceParams.c` update the default ACE configuration stored in *`NADefaultEthInterfaceConfig`* with the settings in `appconf.h`. You must to update these functions to initialize the defaults for the second interface also.

Setting Routing Information

ACE sets the default gateway and subnet mask for the interface, provided it can get this information from either a network server or NVRAM. To set additional routes, use the `ip_add_static_route()` function. The prototype for this function is:

```
int ip_add_static_route(char *device,
                        WORD32 subnetAddress,
                        WORD32 subnetMask,
                        WORD32 gatewayAddress,
                        int cost);
```

Set the arguments as shown here:

Argument	Setting
device	The name of the device.
subnetAddress	The subnet address formatted as a 32-bit word. For example, the IP address 1.2.3.4 would be formatted as 0x01020304.
subnetMask	The subnet mask of the destination network.
gatewayAddress	The address of the gateway for the subnet.
cost	The distance to the subnet in hops.

For example, this code adds a route to the network at `10.64.32.0` with subnet mask `255.255.248.0` and gateway address `10.32.32.1`:

```
ip_add_static_route("eth1", 0x0a40200, 0xfffff000, 0xa202001, 1);
```

Set a default route by passing the *subnetAddress* and *subnetMask* arguments as 0. For example:

```
ip_add_static_route("eth1", 0x0, 0x0, 0xa202001, 1);
```

You can specify only a single default route.

The ip_del_static_route() function deletes static routes. The prototype is:

```
int ip_del_static_route(char *device,
                        WORD32 subnetAddress,
                        WORD32 subnetMask);
```

Set the arguments as shown here:

Argument	Set to
device	The name of the device
subnetAddress	The subnet address to delete
subnetMask	The subnet mask

For example, this code deletes the default route to the subnet at 10.64.32.0:

```
ip_del_static_route("eth1", 0x0a40200, 0xfffff000);
```

Convert the TCP/IP Stack to use the Low Interrupt Latency Option

The TCP/IP stack makes Thread-X kernel calls. Adding additional Ethernet interfaces can cause instabilities if high priority interrupts are prevented from executing.

To avoid this condition, the TCP/IP stack should be configured to use the low interrupt latency option.

To configure the TCP/IP stack for low interrupt latency:

1. In bsp.h, change the BSP_LOW_INTERRUPT_LATENCY definition to TRUE.
2. Recompile the platform's BSP.