V2.10 Migration Guide

Micriμm
For the Way Engineers Work
Weston, FL 33326

Micriμm
1290 Weston Road, Suite 306
Weston, FL 33326
USA
www.micrium.com

Micriμm

For the Way Engineers Work

µC/TCP-IP V2.10 Migration Guide

µC/TCP-IP V2.10 introduces a few new changes in the configuration of devices/interfaces; the API of device driver BSP functions, network buffer functions, and a few constants; the configuration of the loopback interface; and a few new application functions to help improve throughput.

1-0 DEVICE/INTERFACE CONFIGURATION

Device/interface configuration structures were modified to add configurable data area offsets and optional configuration flags. See template device configuration file (`\uC-TCPIP-V2\Cfg\Template\net_dev_cfg.c`) for more information on the changes to configuration structures (`NET_DEV_CFG_ETHER` and `NET_IF_CFG_LOOPBACK`).

First, two network buffer data offsets were added for devices or drivers that require additional octets to be reserved prior to actual receive or transmit packets. Most devices/drivers don't require any offset and should configure both offsets with a value of 0. However, some devices/drivers may require additional octets to be prepared or reserved prior to the Ethernet packet. For these devices/drivers, the receive or transmit offsets should be configured with the required number of additional octets. For example, the Luminary Micro LM3S9B*xx* device prefixes each packet with its frame length. Thus, both the receive and transmit offsets need to be configured with a value of 2 in order to reserve space for the frame length.

Second, all Boolean flags/options were grouped into a single bit-flag configuration. The following bit-field flags may be logically OR'd to enable/select desired options:

`NET_DEV_CFG_FLAG_NONE`	No device configuration flags selected
`NET_DEV_CFG_FLAG_SWAP_OCTETS`	Swap data octets (*i.e.*, swap data words' high-order octet(s) with data words' low-order octet(s), and vice-versa) if required by device-to-CPU data bus wiring and/or CPU endian word order. This option replaces `NET_DEV_CFG_ETHER`'s deprecated `DataOctetSwap` configuration flag.

These optional flags are not required for most devices/drivers and should usually be configured to `NET_DEV_CFG_FLAG_NONE`. In fact, the only flag currently supported, `NET_DEV_CFG_FLAG_SWAP_OCTETS`, is currently used only by the Davicom DM9000 device driver.

1-1 DEVICE DRIVER API CHANGES

`NetBuf_GetDataPtr()` was modified to return the interface's configured network buffer receive/transmit index offsets via the `pix_offset` argument:

```
CPU_INT08U  *NetBuf_GetDataPtr(NET_IF           *pif,
                               NET_TRANSACTION   transaction,
                               NET_BUF_SIZE      size,
                               NET_BUF_SIZE      ix,
                               NET_BUF_SIZE     *pix_offset,
                               NET_BUF_SIZE     *p_data_size,
                               NET_TYPE         *ptype,
                               NET_ERR          *perr);
```

This change is significant for device drivers whose receive and initialization functions configure descriptors with pointers to network buffer data areas:

```
pbuf_desc = NetBuf_GetDataPtr((NET_IF        *) pif,
                              (NET_TRANSACTION) NET_TRANSACTION_RX,
                              (NET_BUF_SIZE   ) NET_IF_ETHER_FRAME_MAX_SIZE,
                              (NET_BUF_SIZE   ) NET_IF_IX_RX,
#if (NET_VERSION >= 21000u)
                              (NET_BUF_SIZE  *)&ix_offset,
#endif
                              (NET_BUF_SIZE  *)&buf_size,
                              (NET_TYPE      *)&buf_type,
                              (NET_ERR       *) perr);
```

Also note that network interface receive buffer index `NET_IF_RX_IX` has been renamed to `NET_IF_IX_RX`. (However, `NET_IF_RX_IX` is still defined to maintain backward-compatibility.)

1-2 LOOPBACK INTERFACE CONFIGURATION

µC/TCP-IP V2.10's loopback interface no longer requires a task nor an RTOS-specific queue. Therefore, configuration of the loopback interface's task and queue is deprecated and may be omitted from `app_cfg.h`. However, the loopback interface still requires configuration via the `NetIF_Cfg_Loopback[]` structure. See template device configuration file (`\uC-TCPIP-V2\Cfg\Template\net_dev_cfg.c`) for more information on loopback interface configuration (`NET_IF_CFG_LOOPBACK NetIF_Cfg_Loopback[]`).

2-0 DEVICE DRIVER BOARD SUPPORT PACKAGE (BSP) CHANGES

Prior to µC/TCP-IP V2.10, all network device BSP functions were global functions called by each device driver passing its interface number via the `if_nbr` argument. The following is an example of how the STM32F107 configured its interrupts prior to V2.10:

```
#if (NET_VERSION < 21000u)
void  NetDev_CfgIntCtrl (NET_IF_NBR   if_nbr,
                         NET_ERR     *perr)
{
    switch (if_nbr) {
        case 1:       /* Configure STM32F107's IF #1 interrupt vector. */
             BSP_IntVectSet(BSP_INT_ID_ETH, NetDev_ISR_HandlerSTM32F107);
             BSP_IntEn(BSP_INT_ID_ETH);
             break;


        case 0:
        default:
            *perr = NET_DEV_ERR_INVALID_IF;
             return;
    }
   *perr = NET_DEV_ERR_NONE;
}
#endif
```

The obvious drawback of this method is that each device's interface number needed to be known at compile-time. Thus, each device needed to be added by the application via `NetIF_Add()` in a specific, known order. And therefore devices could *not* be added dynamically in any random order because then their interface numbers wouldn't correspond with the hard-coded interface numbers in the global device BSP functions.

µC/TCP-IP V2.10 resolves this problem by replacing the global device BSP functions with a set of BSP functions for each device/interface:

```
void  NetDev_CfgClk<DeviceNumber>    (NET_IF   *pif,
                                      NET_ERR  *perr);
void  NetDev_CfgIntCtrl<DeviceNumber>(NET_IF   *pif,
                                      NET_ERR  *perr);
void  NetDev_CfgGPIO_<DeviceNumber>  (NET_IF   *pif,
                                      NET_ERR  *perr);
CPU_INT32U  NetDev_ClkFreqGet<DeviceNumber>(NET_IF   *pif,
                                            NET_ERR  *perr);
```

where **<DeviceNumber>** is the (optional) device number for each specific instance of a device on a development board; optional if the development board does *not* support multiple instances of the device.

Each of these device BSP functions are passed a pointer to the device's network interface structure but does *not* assume any hard-coded interface numbers. This allows the application to dynamically add devices in any order without having to synchronize interface numbers in each device's BSP functions. The following is an example of how the V2.10 STM32F107 device driver can now configure its interrupts:

```
#if (NET_VERSION >= 21000u)
static  void  NetDev_CfgIntCtrl (NET_IF   *pif,
                                 NET_ERR  *perr)
{
    STM32F107_IF_Nbr = pif->Nbr; /* Configure STM32F107's specific IF #.    */
                                 /* Configure STM32F107's interrupt vector. */
    BSP_IntVectSet(BSP_INT_ID_ETH, NetDev_ISR_HandlerSTM32F107);
    BSP_IntEn(BSP_INT_ID_ETH);
   *perr = NET_DEV_ERR_NONE;
}
#endif
```

Note that even though devices may now be added dynamically, the device's interface number must be saved to in order for the device's ISR handler to call `NetIF_ISR_Handler()` with the appropriate interface number. For this example, the device's interface number is saved to global variable, `STM32F107_IF_Nbr`, referenced in the device's ISR function, `NetDev_ISR_HandlerSTM32F107()`:

```
static  void  NetDev_ISR_HandlerSTM32F107 (void)
{
    NET_ERR  err;

    NetIF_ISR_Handler(STM32F107_IF_Nbr, NET_DEV_ISR_TYPE_UNKNOWN, &err);
}
```

However, for most other device BSP functions, the device's interface number is irrelevant and can be ignored. Note that in the following example of how the AT91RM9200 device driver configures its clock, the interface number and interface data structure are not needed:

```
#if (NET_VERSION >= 21000u)
static  void  NetDev_CfgClk (NET_IF   *pif,
                             NET_ERR  *perr)
{
   (void)&pif;                          /* Prevent 'variable unused' compiler warning. */
   *AT91C_PMC_PCER = DEF_BIT_24;        /* Enable EMAC Peripheral Clock.               */
   *perr           = NET_DEV_ERR_NONE;
}
#endif
```

Once each device's BSP functions have been created, a pointer to each BSP function must be added into a BSP interface structure. This allows each device's specific BSP functions to be called by the device driver via pointers to functions. This is similar to how each Ethernet device is called by the Ethernet layer via pointers to API functions. The following is an example of an STM32F107 device BSP interface structure:

```
#if (NET_VERSION >= 21000u)                         /* STM32F107 BSP fnct ptrs : */
const  NET_DEV_BSP_ETHER  NetDev_BSP_STM32F107 = {
    NetDev_CfgClk,                                  /*   Cfg clk(s)              */
    NetDev_CfgIntCtrl,                              /*   Cfg int ctrl(s)         */
    NetDev_CfgGPIO,                                 /*   Cfg GPIO                */
    NetDev_ClkFreqGet                               /*   Get clk freq            */
};
#endif
```

Although certain device drivers may *not* need to call every BSP function, we don't recommend defining any of the device BSP function pointers to `NULL`. Instead, for any unnecessary device BSP function(s), create function pointer(s) to an empty function that returns `NET_DEV_ERR_NONE`. See BSP template file (`\uC-TCPIP-V2\BSP\Template\net_bsp.c`) for examples and notes on how to configure an Ethernet device's BSP functions and BSP interface structure (`NET_DEV_BSP_ETHER`).

Lastly, `NetIF_Add()` was modified to configure each device's BSP functions via the `dev_bsp` argument:

```
NET_IF_NBR  NetIF_Add(void     *if_api,
                      void     *dev_api,
                      void     *dev_bsp,
                      void     *dev_cfg,
                      void     *phy_api,
                      void     *phy_cfg,
                      NET_ERR  *perr);
```

The following example shows how an application could add an STM32F107 device:

```
if_nbr = NetIF_Add((void    *)&NetIF_API_Ether,       /* Ethernet  IF  API. */
                   (void    *)&NetDev_API_STM32F107,  /* STM32F107 dev API. */
#if (NET_VERSION >= 21000u)
                   (void    *)&NetDev_BSP_STM32F107,  /* STM32F107 dev BSP. */
#endif
                   (void    *)&NetDev_Cfg_STM32F107,  /* STM32F107 dev cfg. */
                   (void    *)&NetPhy_API_Generic,    /* Generic   phy API. */
                   (void    *)&NetPhy_Cfg_Generic,    /* Generic   phy cfg. */
                   (NET_ERR *)&err);
```

3-0 APPLICATION DATA ALIGNMENT FUNCTIONS

In order to achieve the best memory copy performance between application data buffers and network buffer data areas, data must be copied to and from application buffers starting at an address that is aligned with network buffer data areas. The following application functions have been added to improve throughput when reading and writing data from network buffers:

```
void  *NetIF_GetRxDataAlignPtr (NET_IF_NBR   if_nbr,
                                void        *p_data,
                                NET_ERR     *perr);
void  *NetIF_GetTxDataAlignPtr (NET_IF_NBR   if_nbr,
                                void        *p_data,
                                NET_ERR     *perr);
```

`NetIF_GetRxDataAlignPtr()` returns the first aligned address in an application data buffer used to read data from network buffers; `NetIF_GetTxDataAlignPtr()` returns the first aligned address in an application data buffer used to write data into network buffers. Since the first aligned address in an application data buffer may be 1 to (`CPU_CFG_DATA_SIZE - 1`) octets from the buffer's starting address, the application data buffer *must* be increased by (at least) an additional (`CPU_CFG_DATA_SIZE - 1`) octets. See network interface source file (`\uC-TCPIP-V2\IF\net_if.c`) for more notes on `NetIF_GetRxDataAlignPtr()` and `NetIF_GetTxDataAlignPtr()`.

The following is an example of sending data through a socket from an aligned application data buffer:

```
CPU_INT08U         AppDataBuf[APP_DATA_SIZE_MAX + CPU_CFG_DATA_SIZE];
CPU_INT08U        *AppDataBufAlignPtr;
CPU_INT16U         AppDataBufSize;
NET_SOCK_RTN_CODE  AppDataBufSizeTxd;
NET_SOCK_ID        AppSockID;
NET_IF_NBR         AppIF_Nbr;
NET_ERR            NetErr;


AppIF_Nbr          = Set app's IF Nbr;
AppDataBufAlignPtr = (CPU_INT08U *)NetIF_GetTxDataAlignPtr((NET_IF_NBR) AppIF_Nbr,
                                                           (void     *)&AppDataBuf[0],
                                                           (NET_ERR  *)&NetErr);
if (NetErr == NET_IF_ERR_NONE) {
    Prepare tx app data (up to APP_DATA_SIZE_MAX) starting @ AppDataBufAlignPtr;
    AppDataBufSize    = Set app data size;

    AppSockID         = Set app sock ID;
    AppDataBufSizeTxd = NetSock_TxData((NET_SOCK_ID) AppSockID,
                                       (void      *) AppDataBufAlignPtr,
                                       (CPU_INT16U ) AppDataBufSize,
                                       (CPU_INT16S ) NET_SOCK_FLAG_NONE,
                                       (NET_ERR   *)&NetErr;
    Handle NetErr conditions;
}
```