Embedding it better...

µTasker Document

µTasker – Networking

Table of Contents

1. Introduction

The µTasker project includes an integrated TCP/IP stack that supports various services that operate over IPv4 and IPv6. Often this operates on a single Ethernet interface to give the device IP connectivity.

The µTasker project is however capable of performing flexible networking which can be one or more of the following combinations:

- A single interface can operate on multiple networks, each with its own IP address
- A single interface can operate on multiple VLANs, each with its own IP address or a mixture of IP and VLAN IDs
- Multiple interfaces can be used with the same capabilities as the single interfaces. This allows multiple physical Ethernet connections or a mixture of Ethernet and other media, such as ZigBee, WiFi and other types that can support IP frames
- Multiple interfaces can share the available network. For example, if there are 5 interfaces 1 and 2 could belong to network 1, interfaces 3 and 4 could belong to network 2 and interface 5 could belong to both networks. Furthermore, one or more of these interfaces could also belong to a further network as well as have varying VLAN memberships

This networking capability allows for great flexibility in performing functions such as bridging between interfaces, filtering traffic or routing between networks.

This document describes how a project can be configured to make use of these advanced features and how the operation is achieved. It illustrates the simple technique which allows the IP services to manage much of the operation autonomously and how it is also fully controlled at the application layers.

2. uSockets

The principle behind the networking operation is the use of the uTasker sockets, known as uSockets. Every UDP and TCP socket is a uSocket, which is a simple identification of the UDP or TCP socket used for individual connection on specific services. For example, a web server may have multiple uSockets listening on the HTTP port 80. Each time a remote client establishes a connection to the web server it uses a free listening socket for the connection, whereby the connection is usually closed once the browser data has been served. Multiple uSockets may be in operation at the same time as the HTTP client requests multiple contents in parallel, or when multiple HTTP clients are browsing the web server at the same time.

Although not all IP traffic is socket oriented, other services such as ICMP ping utility, use "virtual sockets" to manage their operation as will be discussed later. Since all IP services use uSockets (some virtually) the uSocket was chosen as a basis for the management of multiple networks, interfaces and VLANs. The result is very high backward compatibility and simple upward compatibility at all layers, including the application interface. Code efficiency in terms of source code and memory utilisation is also very good with only minimum increases in both, even in the most complicated configurations.

A socket is simply a reference which is returned when a socket is allocated, for example by using the routine `fnGetTCP_Socket()`:

```
USOCKET Telnet_socket = fnGetTCP_Socket(TOS_MINIMISE_DELAY,
usIdleTimeout, fnTELNETListener);
```

Also `fnGetUDP_socket()` returns a socket reference.

The first socket will have the value 0, the next 1, etc. TCP sockets start at 0, as do UDP sockets. That means that there can be UDP and TCP sockets with the value 0 at the same time, whereby their context of use will be clear in the code so they will never be mixed up.

The `USOCKET` can be a `signed char`, a `signed short` or a `signed long`, depending on how many sockets the system will require. A negative value indicates an invalid socket. Its `typedef` is set accordingly in `types.h`.

The following diagram shows how the `USOCKET` is shared for being used as a socket reference plus controlling network, interface and VLAN membership. This is a typical extract from `types.h` and shows the configuration for a system with up to 4 networks, 5 interfaces and up to 4 VLANs. All extensions are optional so only one option, such as sharing two networks on a single interface, could be used, or no such options – or any possible combination of these. Although 4 networks are possible only 2 or 3 of them need to be actually used:

```
typedef signed short         USOCKET;      // 16 bits
#define SOCKET_NUMBER_MASK   0x1f          // 32 sockets (each UDP/TCP)
#define VLAN_SHIFT           5
#define VLAN_MASK            0x3           // up to 3 alternative VLANs
                                           // (max. 15 - limit 0xf)
#define INTERFACE_SHIFT      7
#define INTERFACE_MASK       0x1f          // 5 physical interfaces
#define USER_INFO_SHIFT      12
#define USER_INFO_MASK       0x1           // valid for 2 user functions
#define NETWORK_SHIFT        13
#define NETWORK_MASK         0x3           // up to 4 networks


/* 15  14   13   12   11   10    9    8    7    6    5    4   3   2   1   0 */
/*-------------------------------------------------------------------------*/
/* V  | N1 | N0 | U0 | I4 | I3 | I2 | I1 | I0 | V1 | V0 | S | S | S | S | S */
/*-------------------------------------------------------------------------*/

// V = valid socket
// N = Network (0 / 1 / 2 / 3) that the socket can use
// U = User (0 / 1) - optional
// Ix = flags for each interface that the socket can use
// (I0, I1, I2, I3 and I4 means that there are 5 physical sockets available)
// Vx = virtual lan membership reference - optional
// (0 is standard virtual LAN and 1,2,3 are alternative VLANs)
// S = socket number from 0..0x1f (maximum 32 TCP and 32 UDP sockets possible)
// note that USOCKET has been chosen as signed short to give adequate width
```

Depending on the system's requirements the dimensioning of the fields for network (N), Interface (I), VLAN (V) and User (U), as well as the width of the storage type can be configured accordingly. If the sockets are uses on a single network and single interface (eg. a single Ethernet connection with a single IPv4 address) with or without a single VLAN there is no configuration requirement (the system defaults to the simple configuration).

The exact number of networks, interfaces, users and VLANs is defined by `IP_NETWORK_COUNT` and `IP_INTERFACE_COUNT`. Eg.

```
#define IP_INTERFACE_COUNT          5
#define IP_NETWORK_COUNT            5
```

If a particular define is missing the system defaults to a single network/interface.

If VLAN is used the define `SUPPORT_VLAN` needs to be set. This enabled basic VLAN operation where the system operates on a single VLAN ID, which can be enabled and disabled by the software as required.

The define `SUPPORT_DYNAMIC_VLAN` allows frames with non-matching VLAN tags to be handled. Rather than rejecting all VLAN tagged receptions that don't match the VLAN ID the frames are tagged according to their VLAN properties and passed on for handling as appropriate by the application, which can reject or redefine VLAN memberships as it wishes. The application must supply the routine

```
extern int fnVLAN_Accept(ETHERNET_FRAME *ptr_rx_frame);
```

where individual decisions can be made.

The define `SUPPORT_DYNAMIC_VLAN_TX` allows additional transmission control of VLAN frames. The application must supply the routine

```
extern int fnVLAN_tag_tx(QUEUE_HANDLE channel);
```

where individual decisions can be made.

What using multiple networks, each received frame needs to be associated to one of the available networks. The application must supply the function

```
extern unsigned char fnAssignNetwork(ETHERNET_FRAME *frame, unsigned
char *ptrDestinationIP)
```

which assigns the received frame to a particular network to be handled on. It can return `DEFAULT_NETWORK` for the standard one or else any other available one. The decision is usually made based on the destination IP address. The function is called after VLAN handling has taken place (if used) so that VLAN decisions can be used as input to the network selection process if required. This function is called when ARP or IP frames are received and so the pointer to the destination IP address is to the corresponding content of the particular frame. The `ETHERNET_FRAME` pointer is also passed for advanced decisions in case the frame type needs to be checked and taken into account.

Once this configuration is made, the advanced operation features are available.

3. Compatibility when using Advanced Networking Option

When no advanced options are enabled code is backward compatible as long as the project version includes multi-homed/multi-interface support. If an older project version needs to be used there are four simple modifications that are required which are described in Appendix A.

When using the extended optional capabilities there are a few small adjustments that need to be made when calling functions (to make use of the capabilities) and generally when handling sockets in listener call-backs and when ARP messages inform of ARP resolution results. These are discussed in more detail in the following sections.

4. Working with TCP Sockets

When TCP sockets are allocated the `USOCKET` reference includes no extended details. When the `USOCKET` is used as returned by the `fnGetTCP_Socket()` function it will cause all operation to be on the first interface and the first network (default network and default interface). This also guaranties basic backward compatibility when the new capabilities are activated on the default network and interface.

A few examples will now be given to illustrate how the extended `USOCKET` details are used to control operation. Each example assumes a system with 3 interfaces on which IP traffic can be received/transmitted on [Interface 1, Interface 2 and Interface 3]. Interfaces 1 and 2 have an IP configuration belonging to the default network – Network 1, and Interfaces 2 and 3 have an IP configuration belonging to Network 2. Note that Interfaces 1 and 3 belong to different networks but Interface 2 has two IP addresses and belongs to both networks.

Although VLAN operation is not discussed in these sections, the VLAN details are also contained in the `USOCKET` as discussed previously.

Example 1: Handling connections to a TCP server that arrive on multiple networks and/or multiple ports

Each TCP server will have its own TCP socket (`USOCKET`). The allocated socket will have no network/interface details. When an interface receives a frame belonging to its network it adds the network and interface information to the received `USOCKET` which is passed to the listener:

```
USOCKET myTCPSocket;  // USOCKET allocated for use by TCP listener
...

static int fnListener(USOCKET Socket, unsigned char ucEvent, unsigned char
*ucIp_Data, unsigned short usPortLen)
{
    if (_TCP_SOCKET_MASK(myTCPSocket) == _TCP_SOCKET_MASK(Socket)){
        switch (ucEvent) {
        case TCP_EVENT_CONREQ:                             // session requested
            ...
            myTCPSocket = Socket;    // add network, VLAN and interface details
        break;
        ...
        }
        return APP_ACCEPT;
    }
    return APP_REJECT;
}
```

This code shows a typical TCP server's listener call-back, whereby two lines a of code are highlighted which are important for its correct operation.

First of all, the `USOCKET` value passed can contain additional information and so, when comparing the validity of the `USOCKET` value, the macro `_TCP_SOCKET_MASK(myTCPSocket)` should be used to extract only the `USOCKET` reference part for the comparison (receptions may be arriving on different interfaces, for instance). This macro can be left in code even when the system uses no extension options since it equates to a standard compare of two `USOCKET` values with no additional code overhead.

The second point to note is that, while the TCP connection exists the `USOCKET` value used by the TCP interface includes the additional network/interface information. Therefore the TCP sockets value is updated to this value when the TCP connection starts. The local `USOCKET` therefore contains the same information which is valid while the connection is active. Any further actions, such as sending data over the open TCP connection then automatically takes place on the correct network and interface. For example:

```
fnSendTCP(myTCPSocket, tcp_data_frame, tx_length, 1);
```

It is not necessary to reset the information when the connection closes since the socket reference parts always remains the same.

Example 2: Connecting to a remote TCP server

The following call can be used to establish a connection to a remote TCP server:

```
fnTCP_Connect(myTCPSocket, ucIP_address, usPortNumber, 0, 0);
```

Since our system has three interfaces, whereby one of them belongs to two different networks, the question is over which network and on which interface(s) should the connection be made on? Possibly the caller doesn't have enough information to make this decision and so it should just be connected on the appropriate interface. In any case, the caller can decide on the scope of the connection by defining the USOCKET extension before it use as appropriate.

If we know that the destination IP address is on Network 2 but we don't know whether it is reachable on Interface 2 or Interface 2 we can do the following:

```
_TCP_SOCKET_MASK_ASSIGN (myTCPSocket); // reset any previous extension details
addNetworkInterface(myTCPSocket, NETWORK2, (INTERFACE2 | INTERFACE3));
fnTCP_Connect(myTCPSocket, ucIP_address, usPortNumber, 0, 0);
```

This configures the `USOCKET` to cause the connection to be made (only) on network 2 and allow interfaces 2 or 3 to be used.

Defines for the networks and interface could be

```
#define NETWORK2    DEFAULT_NETWORK // 0

#define NETWORK2    1

#define INTERFACE1  DEFAULT_INTERFACE // 0

#define INTERFACE2  1

#define INTERFACE3  2
```

Although the networks and interfaces could also be given more meaningful defines in particular projects, such as

```
#define WIFI_INTERFACE 1

#define ZIGBEE_INTERFACE 2
```

The fact that two interfaces are defined for the connection use allows ARP to try to resolve the destination address on both interfaces (if the IP address is already in ARP cache the passed interface values are not used since the corresponding interface will automatically be used).

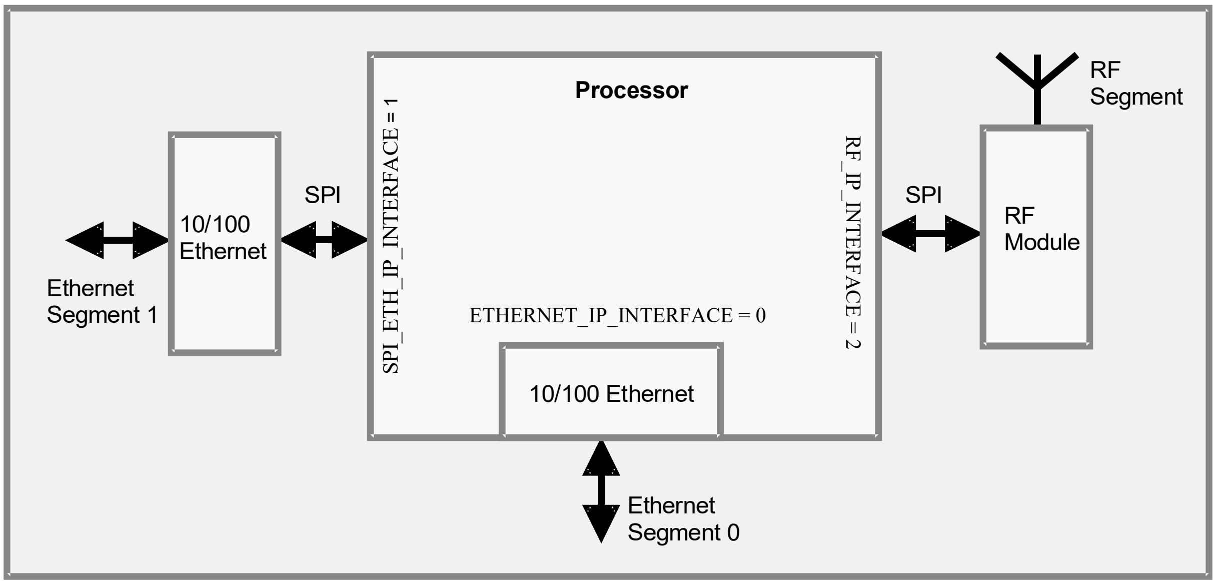
If the destination is not to be resolved by ARP on `INTERFACE3`, setting just `INTERFACE2` will restrict ARP to using only this one when resolving destinations.

Note that the macro `addNetworkInterface()` is used to add the details to the `USOCKET`. This can also be left in code when the extended operation is disabled since the macro will then equate to an empty line of code.

Furthermore, the ARP cache saves the interface value (with the handle belonging to the interface). This allows the interface used to reach the destination IP to be monitored (for example with `fnGetARPentry()`).

5. Practical Examples

The first example discusses the operation of a board on a single network but with three interfaces. The first interface is the processor's internal Ethernet controller. The second interface is an SPI based Ethernet controller which has been connected to an SPI module in order to extend the Ethernet interfaces to two physical ones. The third interface is an RF interface, also connected to an SPI module of the processor, adding a third wireless Ethernet type interface to the board.



The interfaces are assigned unique numbers 0, 1 and 2 (when there is only one interface this has always the number 0 – *the default interface*).

In this case each of the interfaces are on different network segments (eg. Ethernet 0 and Ethernet 1 segments are not directly connected in any way and any traffic between the two can only pass via the processor. The same is also true for the RF segment. In this case it is possible to use a single MAC address shared by the three interfaces.

Due to the fact that there are three individual interfaces that need to be uniquely identified, the `USOCKET` strategy foresees 3 interface bits.

```
    #define INTERFACE_SHIFT  7
    #define INTERFACE_MASK   0x07 // three interfaces supported,
```

whereby the INTERFACE_MASK is important since the INTERFACE_SHIFT value simply depends on where the bits are to be located in the available USOCKET bit space.

IP_INTERFACE_COUNT is of course set to 3.

The three interfaces are given names according to their interface numbering:

```
#define ETHERNET_IP_INTERFACE    DEFAULT_IP_INTERFACE // 0
#define SPI_ETH_IP_INTERFACE    (unsigned char)1
#define RF_IP_INTERFACE         (unsigned char)2
```

noting that the default IP interface is 0.

All interfaces share the Ethernet method which is based on an Ethernet handle. The handle is obtained by calling the open function (*following examples are based on the RF interface*):

```
RF_Handle = fnOpen(TYPE_ETHERNET, FOR_I_O, &RF_ethernet);
```

whereby `RF_ethernet` includes information about the configuration including the interface number (passed as `ppp_ethernet.Channel`) and notably a list of call-back functions that are to be used by the generic Ethernet driver to interact with the interface's implementation. These are

```
typedef struct stETHERNET_FUNCTIONS {
    int (*fnConfigEthernet)(ETHTABLE *); // configuration function for the Ethernet interface
    int (*fnGetQuantityRxBuf)(void); // call-back used to get the number of available receive buffers
    unsigned char *(*fnGetTxBufferAdd)(int); // call-back used to get a memory-mapped buffer address
    int (*fnWaitTxFree)(void);       // call-back used to allow waiting on transmit buffer availability
    void (*fnPutInBuffer)(unsigned char *, unsigned char *, QUEUE_TRANSFER);
                                     // call-back used to prepare transmit data to the output buffer
    QUEUE_TRANSFER (*fnStartEthTx)(QUEUE_TRANSFER, unsigned char *);
                                     // call-back used to release a prepared transmit buffer
    void (*fnFreeEthernetBuffer)(int); // call-back used to free a used reception buffer
#if defined USE_IGMP
    void (*fnModifyMulticastFilter)(QUEUE_TRANSFER, unsigned char *);
                                     // call-back used to setup the multicast filter
#endif
} ETHERNET_FUNCTIONS;
```

whereby some may be implemented as dummy functions when not of relevance for the interface used.

For each of the interfaces the interface's socket <u>bit reference</u> is also defined, which is the one used mainly in software coding (!!)

```
#define ETHERNET_INTERFACE defineInterface(ETHERNET_IP_INTERFACE)
                   // Ethernet interface is default interface
#define SPI_ETH_INTERFACE  defineInterface(SPI_ETH_IP_INTERFACE)
                   // SPI Ethernet interface is second interface
#define RF_INTERFACE       defineInterface(RF_IP_INTERFACE)
                   // RF interface is third interface
```

The significance of the default interface is that it will be used in situations where the involvement of other are not explicitly defined.

So that an interface can be used by TCP/IP it must have an interface handle – this matches the interface to a device driver for the actual transmission and reception of IP frames. In the case of the main Ethernet interface, which is the default one, it is entered when the Ethernet task is started by using:

```
fnEnterInterfaceHandle(DEFAULT_IP_INTERFACE, Ethernet_handle);
```

Each further interface needs to be entered (once) using the same call. This takes place either after the interface's driver has been opened so that its handle is known.

Note: In the case of a single interface being used, the handle is entered as
`Ethernet_handle[0] = RF_Handle;`

As can be seen the the call-back functions, IP datagrams to be sent to the interface are prepared by the functions

```
void (*fnPutInBuffer)(unsigned char *, unsigned char *, QUEUE_TRANSFER);
```

and

```
QUEUE_TRANSFER (*fnStartEthTx)(QUEUE_TRANSFER, unsigned char *);
```

which allows the IP traffic to be handled in a certain way before transmission; for example, coding for compatibility with the physical interfaces (SPI in this case).
Reception is handled by collecting a datagram to be passed to the IP interface (which may also require some form of pre-parsing specific to the interface used) and then passing it as an Ethernet frame using

```
fnHandleEthernetFrame(&rx_frame, RF_Handle);        // handle the reception
```

The following example shows how the data buffer is passed, along with details of the interface and its characteristics that are relevant for the IP handling:

```
ETHERNET_FRAME rx_frame;
rx_frame.frame_size = (unsigned short)rf_frame_length; // length of datagram
rx_frame.ptEth = (ETHERNET_FRAME_CONTENT *)ucRF_InputMessage; // buffer containing the data
rx_frame.usDataLength = 0;
rx_frame.usIPLength = 0;
#if (IP_INTERFACE_COUNT > 1)
rx_frame.ucInterface = (RF_INTERFACE >> INTERFACE_SHIFT); // reception is on the RF interface
rx_frame.ucInterfaceHandling = (DEFAULT_INTERFACE_CHARACTERISTICS | INTERFACE_NO_MAC_FILTERING);
                                                    // default interface handling
#endif
#if defined IPV4_SUPPORT_RX_DEFRAGMENTATION && defined IP_RX_CHECKSUM_OFFLOAD
rx_frame.ucSpecialHandling = (INTERFACE_NO_TX_PAYLOAD_CS_OFFLOADING |
                              INTERFACE_NO_TX_CS_OFFLOADING | INTERFACE_NO_RX_CS_OFFLOADING |
                              INTERFACE_NO_MAC_FILTERING);
#endif
fnHandleEthernetFrame(&rx_frame, PPP_Handle);         // handle the reception
```

Note that an interesting such interface is that used by RNDIS (USB-CDC based Remote Network Distributed Interface Specification) as discussed in http://www.uTasker.com/docs/uTasker/uTaskerRNDIS.pdf.

6. Conclusion

This document has explained the networking concept used in the µTasker project to allow multiple interfaces and networks to be implemented in a simple but powerful fashion based on the `USOCKET` reference.

After explaining the control method that applications can use to determine the network and interface interactions practical examples have been presented to illustrate use cases.

Modifications:

V1.00 10.05.2017: First release

Appendix A – Upgrading Existing Applications

The uTasker project has new support for multi-homed networking. This means that the TCP/IP stack can handle multiple networks and also multiple interface (including multiple interfaces in each network)
This new support required a few adjustments that affect the original definitions.
The following is a step by step explanation concerning making existing projects compatible with the new use if multi-network/interface operation is not required:

1)
Originally `NETWORK_PARAMETERS network`; (and `network_flash`) was a single network `struct` holding the IP defaults of the single network
New: `NETWORK_PARAMETERS network[IP_NETWORK_COUNT];` this should now be defined as an array of network `structs`, whereby `IP_NETWORK_COUNT` defaults to `1`

2)
Originally `const PARS cParameters;` (as well as `TEMPPARS *temp_pars;` and `PARS *parameters;`) had a single entry `usServers` to control the single network's servers operation.
New: `unsigned short usServers[IP_NETWORK_COUNT];` this should now be defined as an array, whereby `IP_NETWORK_COUNT` defaults to `1`

3)
All uses of `network` (and `network_flash`) (point 1) and `PARS` based structs need to be changed to, for example, `temp_pars->temp_parameters.usServers[DEFAULT_NETWORK]`, whereby `DEFAULT_NETWORK` defaults to `0`

4) The interface `fnGetARPentry()` has been changed to be more flexible.

Rather than returning either the IP or MAC address in an ARP cache entry it returns a pointer to the entry so that more information can be displayed.

```
extern unsigned char *fnGetARPentry(unsigned char ucEntry, int
iIP_MAC);
```

has been changed to

```
extern ARP_TAB *fnGetARPentry(unsigned char ucEntry, int iIP_MAC);
```

This means that routines no longer pass the requested information but instead extract the required part from the `ARP_TAB` pointer. For example:

```
cPtr = (CHAR *)fnGetARPentry((unsigned char)(*ptrBuffer - 'a'),
GET_IP); // get pointer to IP address
```

becomes

```
ARP_TAB *PtrEntry = fnGetARPentry((unsigned char)(*ptrBuffer -
'a')); // get pointer to ARP entry
cPtr = PtrEntry->ucIP;
```

The four changes above are adequate to make existing projects fully compatible.

Appendix B – Tail-Tagging with the Micrel KSZ8863

The Micrel KSZ8863 is a two port switch which connects to an Ethernet controller interface via (R)MII interface (as PHY). It supports the standard MDIO communication for trivial settings and monitoring or a SMI interface, sharing the same lines but not fully compatible with MDIO. The SMI interface must be used to work with the advanced features of the PHY.

The KSZ8863's tail-tagging mode, which can be enable via SMI, adds a tag to the end of each received Ethernet frame informing on which port (which of the switch interfaces) it was received on. It is also possible for software to add a tag to each Ethernet frame passed to the PHY to control over which of the switch ports it is physically transmitted. The choices are, on a frame-by-frame basis, to send a frame in normal switch mode, to send it to the first switch port, to the second switch port or to BOTH switch ports.

The tail-tagging mode fits well into the uTasker's networking capabilities due to the fact that this opens up the ability to use a singe Ethernet controller in conjunction with the KSZ8863 to realise two physical Ethernet interfaces. These interfaces can then be used for multi-homed networking or various other strategies.