DALLAS SEMICONDUCTOR MAXIM

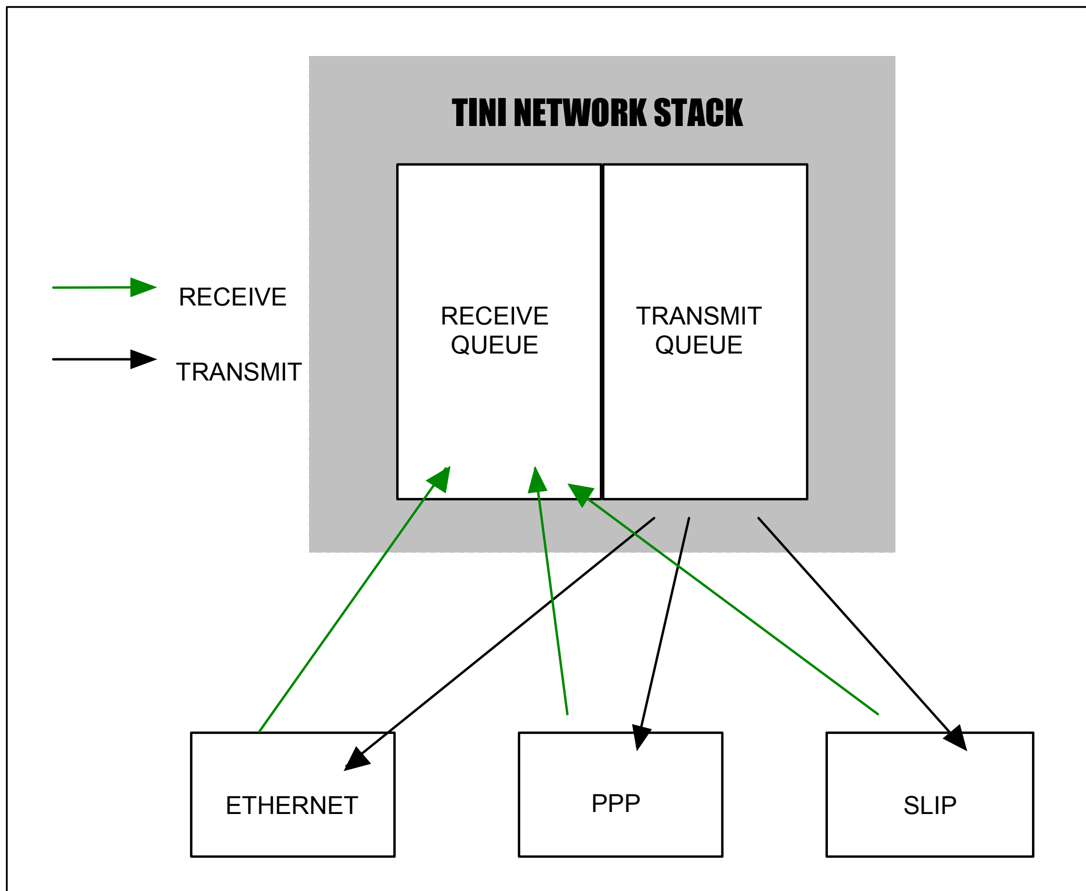
Application Note 2244
Implementing a Network Interface in TINI 1.1x

www.maxim-ic.com

OVERVIEW

Dallas Semiconductor designed the TINI® network stack with support for network interfaces other than Ethernet. One example, the PPP interface, is bundled with the TINI firmware. Until recently, the internals of the network stack have been locked behind the Java™ runtime layer. However, moving PPP into a native library required exposing key network routines to the native interface. In the TINI 1.1x family firmware, it is possible in native to implement additional network interfaces for the TINI network stack. This opens up the possibility of support for IP other physical media. This application note describes the functions and callbacks used to implement a network interface, and provides a sample SLIP interface.

Network Interfaces



TINI is a registered trademark of Dallas Semiconductor.
Java is a trademark of Sun Microsystems.

In concept, a network interface is straightforward. A network interface connects the TCP/IP stack to the physical media. This requires the following:

A local and gateway address
A network mask
A transmit function

The network settings are used to route packets from the stack to the proper interface on transmit. The transmit function takes a packet from the network queue and performs the magic to send it over the physical media. This function must be implemented as a native callback; it cannot be written in Java. As packets are received over the physical media, the network interface must pass them to the network stack receive queue.

Kernel Memory and Network Queues

One complication is that TCP packets are both out of order and variable length. This requires variable length blocks of data to be queued and reordered. However, allocation of variable length blocks from the heap can be at best inefficient, and at worst a serious performance drag.

In order to speed up memory operations with TCP packets, TINI has a set of pre-allocated buffers specifically for network operations called *kernel buffers.* Kernel buffers are recycled by all network interfaces, so they don't have the cost of allocating from the heap. However, there are a few drawbacks.

> Kernel buffers are not guaranteed to be available. If a large number of packets are on the queue, then there is a higher probability that you won't be able to get a kernel buffer, especially if you have a larger packet. In this case you will have to drop the packet.
> Memory leaks with kernel buffers are especially dangerous, and can quickly crash the system. Great care must be taken to ensure releasing kernel buffers when you are finished with them.

The network interface transmit function is passed a kernel buffer containing the packet to transmit. It should not physically transmit the packet when it is called. Instead, it should add the packet to a queue to be transmitted at a later time, in either a thread, inside of a poll function, or in an interrupt. TINIOS has a set of functions for handling the queuing of the kernel buffers.

Introduction to SLIP

According to lore, the original design for SLIP was written on a napkin. True or not, SLIP was a dominant IP over serial protocol until PPP took the crown. It is a straightforward protocol with very little processor overhead.

SLIP is only slightly more complicated than sending packets raw over serial. A magic character is defined to denote the end of a packet. As that character might appear in the data stream, it is escaped out of the data using a special escape character. As the escape character might appear in the data stream, a double escape denotes a regular escape character.

Basic SLIP does not support authentication, IP address assignment, or other nifty features. This makes it an ideal example for a network interface.

The Native Library

The native library handles the low level SLIP work. As a warning: it is a rather advanced native library, using the I/O, network, and thread functions directly. If you are not familiar with TINIOS native programming, there are better introductions than this.

The initialization function allocates the memory and indirects used by the library. It then allocates a network queue and stores the handle at the beginning of the ephemeral state block. This will be used by the transmit function for queuing outbound packets.

The `slip_init` function opens the serial ports in native code. This gives the native code access to the serial handles, so that transmission can be handled in the native layer.

`slip_ioctl` is the multipurpose connection between our interface and Java. Commands are sent to the interface using this function.

`slip_transmit` is the function that handles transmit requests from the network stack. It takes the incoming packet and adds it to the transmit queue. Note that the SLIP implementation does not transmit any data from this function, as this would steal time from network processing. Instead, it places the data into a transmit queue, for delayed transmission.

There are two loops that are executed in separate Java threads: `slip_read_loop` and `slip_write_loop`. These two functions handle the majority of the slip implementation.

slip_read_loop reads incoming packets in a loop. It calls slip_read_packet, which reads in the packet byte by byte from the serial port, handling the escaping as it is read in. Once a packet is received, it is verified to be valid. If it passes that test, a kernel buffer is allocated, and the packet is copied into the kernel buffer. Finally, the packet is passed to the network stack, and we loop again. If no data is pending, read will suspend, allowing other threads to run.

`slip_write_loop` pulls a packet off the transmit queue. If no packets are pending, it yields to other threads. Otherwise, a packet is dequeued and SLIP encoded into a static buffer. The kernel buffer is then freed, and the packet is written to the serial port. The process then begins again.

The com.dalsemi.tininet.slip.Slip Class

A simple Java network wrapper is provided. This handles loading the network library, adding the interface, and providing methods for the network settings.

Adding the network interface to TINI is done in the `addInterface` method with the following code:

```
TININet.addInterfaceEntry(name.getBytes(), // Interface Name
      localAddr,      // Local address
      netMask,        // Network mask
      remoteAddr,     // Gateway
      (byte)0x43,     // ACTIVE | DEFAULT
      (byte)0x02,     // User defined
      slipIoctl(IOCTL_INTERFACE,0), // Transmit function
      256,             // Max transmission unit
      TININet.TCP_TIMEOUT_128); // Timeout
```

TININet documentation is available in the TINI API javadocs.slipIoctl(IOCTL_INTERFACE,0) returns a pointer to the network interface transmit function.

There is a static main inside of the class that gives an example of how to use it. First, we create an instance of the class.

```
Slip slip = new Slip();
```

Next, we set the network settings. Remember, this is not negotiated by the protocol.

```
System.out.println("Setting address vars");
slip.setLocalAddress(new byte[]{(byte)192,(byte)168,(byte)0,(byte)1});

slip.setRemoteAddress(new byte[]{(byte)192,(byte)168,(byte)0,(byte)2});

slip.setNetMask(new byte[]{(byte)255,(byte)255,(byte)0,(byte)0});
```

Next, we initialize the serial port.

```
CommPortIdentifier cpi = CommPortIdentifier.getPortIdentifier("serial1");
SerialPort sp = (SerialPort)cpi.open("slip",5000);

System.out.println("Setting serial port params");
sp.setSerialPortParams(115200,SerialPort.DATABITS_8,
```

```
        SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
```

Now, lets add the network interface.

```
System.out.println("Adding interface");
slip.addInterface("slip");
```

Finally, we can start the interface. This takes control of the serial port, and starts the slip loops in independent threads.

```
System.out.println("Starting interface");
slip.up(sp);
```

Nothing else for us to do but handle packets.

```
Thread.currentThread().suspend();
```

Conclusion

Network Interfaces allow the TINI network stack to support networking over additional physical media. The TINI 1.1x firmware allows developers to add their own network interfaces to the TINI network stack. This opens new horizons for the TINI platform.

APPENDIX A: ADDITIONAL FUNCTIONS USED BY THE SLIP PROTOCOL

Queue Functions

```
;****************************************************************************
;*
;* Function Name: Network_Q_Create
;*
;*   Description: Create a new queue of specified size.
;*
;*      Input(s): (dptr1) -> location for new queue handle pair
;*                a       -> max element count
;*                b       -> element length
;*
;*    Outputs(s): a -> 0 if successful, OutOfMemoryError otherwise.
;*
;*     Destroyed:
;*
;*         Notes:
;*
;****************************************************************************
;*
;* Function Name: Network_Q_Free
;*
;*   Description: Remove entire queue.
;*
;*      Input(s): (dptr1) -> queue handle pair
;*
;*    Outputs(s): a -> 0 if successful, NullPointerException otherwise.
;*
;*     Destroyed:
;*
;*         Notes:
;*
;****************************************************************************
;*
;* Function Name: Network_Q_Add
;*
;*   Description: Add an element to the specified queue.
;*
;*      Input(s): (dptr1) -> queue handle pointer pair
;*                (dptr2) -> element
;*
;*    Outputs(s): a -> 0 if successful, InternalError if queue is full
;*
;*     Destroyed:
;*
;*         Notes:
;*
;****************************************************************************
;*
;* Function Name: Network_Q_IsFull
;*
;*   Description: Test queue to see if it is full.
;*
;*      Input(s): (dptr1) -> queue handle pair
;*
```

```
;*   Outputs(s): a -> 0 if not full, non-zero otherwise
;*
;*    Destroyed:
;*
;*        Notes:
;*
;*****************************************************************************
;*
;* Function Name: Network_Q_IsEmpty
;*
;*   Description: Test queue to see if it is empty.
;*
;*     Input(s): (dptr1) -> queue handle pair
;*
;*   Outputs(s): a -> element count
;*
;*    Destroyed:
;*
;*        Notes:
;*
;*****************************************************************************
;*
;* Function Name: Network_Q_GetTop
;*
;*   Description: Test for an empty queue. If the queue isn't empty a
;*               pointer to the top element is returned.
;*
;*     Input(s): (dptr1) -> queue handle pointer pair
;*
;*   Outputs(s): (dptr1) -> queue entry
;*               a       -> 0 if empty or not a queue, non-zero otherwise
;*
;*    Destroyed:
;*
;*        Notes: Check of the magic number is supressed.
;*
;*****************************************************************************
;*
;* Function Name: Network_Q_Remove
;*
;*   Description: Remove top ('head') element from the specified queue.
;*
;*     Input(s): (dptr1) -> queue HPP
;*
;*    Output(s): a -> 0 if successful, InternalError if queue is empty
;*
;*    Destroyed:
;*
;*        Notes:
;*
;*****************************************************************************
```

IP Functions

```
;*****************************************************************************
;*
;* Function Name: Network_IP_CheckHeader
;*
;*   Description: Check to see if we're interested in the incoming datagram.
```

```
;*            Also do some checks to verify basic correctness.
;*
;*     Input(s): (dptr0) -> IP header
;*
;*   Outputs(s): a -> 0 if successful, non-zero otherwise
;*            b -> 0 if datagram was directed, non-zero for broadcast/
;*                 multicast
;*
;*    Destroyed:
;*
;*        Notes: Before calling this function, the caller must ensure
;*            1) the buffer contains at least 20+8 (header + protocol)
;*              bytes of IP data
;*            2) the total packet length reported in the IP header
;*              is smaller than or equal to the size of the buffer
;*              (minus frame length)
;*            We don't check the header checksum, since it's been
;*            proven to be useless (or else IPv6 would have one)
;*
;****************************************************************************
;*
;* Function Name: Network_IP_PacketReceived
;*
;*  Description: Called by a network interface driver when an IP datagram
;*            has been received. At least the IP header is assumed to
;*            have been unloaded.
;*
;*     Input(s): (dptr0) -> IP header
;*            r3:r2   -> handle to kernel buffer containing datagram.
;*
;*   Outputs(s):
;*
;*    Destroyed: All network stack registers
;*
;*        Notes: It is assumed that CheckHeader has been called and
;*            returned 0.
;*
;****************************************************************************
```