Freescale Semiconductor
Application Note

Document Number: AN3470
Rev. 0, 06/2007

ColdFire TCP/UDP/IP Stack and RTOS

by Eric Gregori

1 Introduction

Transmission Control Protocol /Internet Protocol (TCP/IP) is the Internet's communication protocol. The acronym is derived from two layers of the communication stack, TCP and IP. The term TCP/IP actually describes multiple protocols in both stacks. Each protocol is defined by a Request For Comment (RFC).

Proper TCP/IP stack operation requires multi-tasking. The ColdFire® TCP/IP stack is integrated with a simple operating system. This simple round-robin OS can also be used by the application code. The OS is non-preemptive, provides two modes of operation (single stack, or superloop, and multi-stack). Additional OS features include a interactive real-time upgradeable menu system, user timers, and heap memory management.

Contents

freescale™ semiconductor

2 ColdFire TCP/UDP/IP Stack Features

- HyperText Transport Protocol (HTTP), Serial to Ethernet, Trivial File Transfer (TFTP)
- Mini IP Application's Interface
- Dynamic Host Configuration Protocol (DHCP) or manual IP configuration, DNS
- Transmission Control Protocol (TCP), User Datagram Protocol (UDP)
- Internet Control Messaging Protocol (ICMP), BOOTP (BOOTstrap Protocol)
- Address Resolution Protocol (ARP), Internet Protocol (IP)

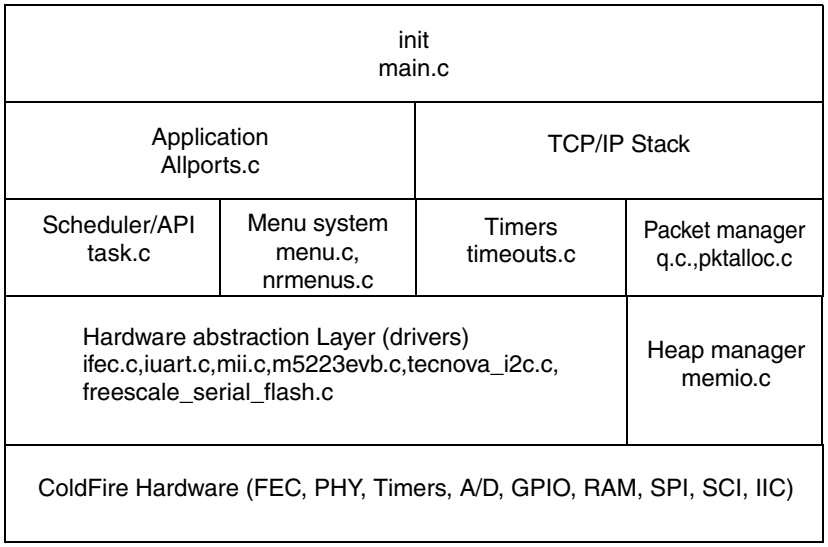


Figure 1. ColdFire TCP/IP Stack and RTOS

3 ColdFire TCP/IP Stack and RTOS

3.1 RTOS Overview

The TCP/IP stack requires that multiple processes or tasks occur simultaneously. This is accomplished using one of two methods: superloop (RTOS disabled) or a multi-tasking operating system (RTOS).

In superloop mode, each task is a function call. All the tasks share a common stack. This mode is the most efficient in terms of memory usage. By using only one stack, less space is wasted. Also, with the RTOS disabled there are no RTOS RAM requirements (TCBs and other structures) using valuable RAM resources. The disadvantage of superloop mode is that it is static. Tasks cannot be created or deleted at runtime. Depending on the architecture/requirements of the system, this may end up using more RAM instead of less.

Sleep is not supported in superloop mode (the RTOS is disabled). Each task function must return to the main loop after its job is done. This requires a very different task architecture then when using the RTOS.

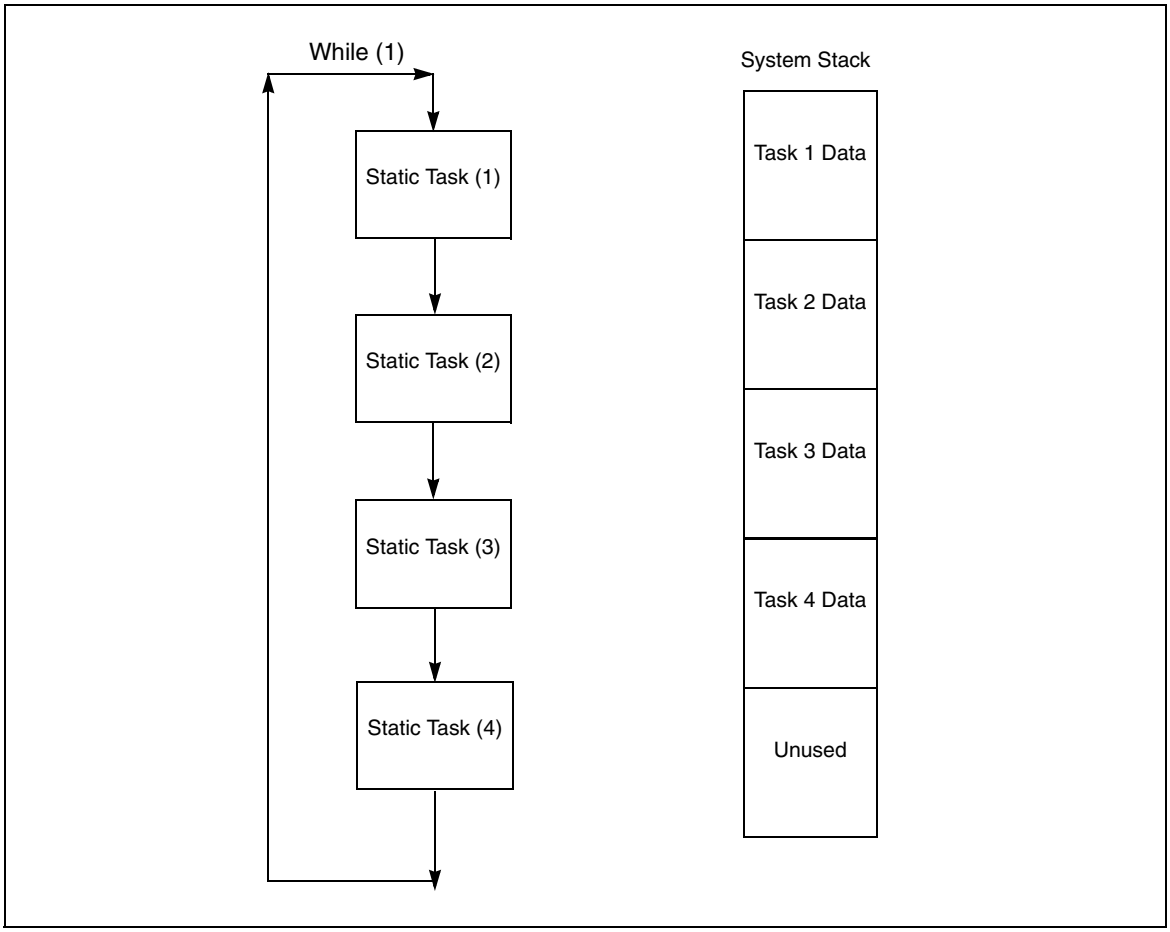


Figure 2. Superloop Structure and Stack Usage

3.1.1 Sample Superloop Code

```
void emg_superloop(void)
{
While(1)
{
                kbdio();                    // Call menu task (function)
                packet_check();             // Call network stack state machine
                inet_timer();               // Check for timer timeouts
                task1();                    // Call user task1
                task2();                    // Call user task2
                task3();                    // Call user task3
                task4():                    // Call user task4
}
```

When the RTOS is enabled, it provides a dynamic mode of operation for the stack. Tasks can be created and destroyed at runtime. When a task is created, a new stack is created for the task by allocating RAM from the heap. When the task is destroyed, the RAM allocated for the task's stack is returned to the heap. In this way, if your application is dynamic, using the RTOS may be more RAM efficient.

When a task is created, memory is allocated from RAM (via the heap) for the new tasks stack. The size of each stack is static, and determined at compile time. The stack size must be big enough to accommodate not only the task needs, but any interrupts used by the system as well.

With the RTOS enabled each task has a task control block. The TCBs are linked by a linked list. The TCB structure is declared in task.h. This simple RTOS does not support task priorities. The scheduler simply increments to the next TCB in the list, executing the task pointed to by the TCB if the task is ready to execute (not sleeping). Since this RTOS is also non-preemptive, task switching only occurs when a task wants to give up control. A task gives up control by calling tk_block() or going to sleep (tk_sleep()).

```
struct task
{
        struct task *tk_next;    /* pointer to next task */
        stack_t          *tk_fp;         /* task's current frame ptr */
        char             *tk_name;       /* the task's name */
        int               tk_flags;      /* flag set if task is scheduled */
        unsigned long       tk_count;      /* number of wakeups */
        stack_t          *tk_guard;      /* pointer to lowest guardword */
        unsigned    tk_size;       /* stack size */
        stack_t          *tk_stack;      /* base of task's stack */
        void             *tk_event;      /* event to wake blocked task */
        unsigned long       tk_waketick;   /* tick to wake up sleeping task */
};
```

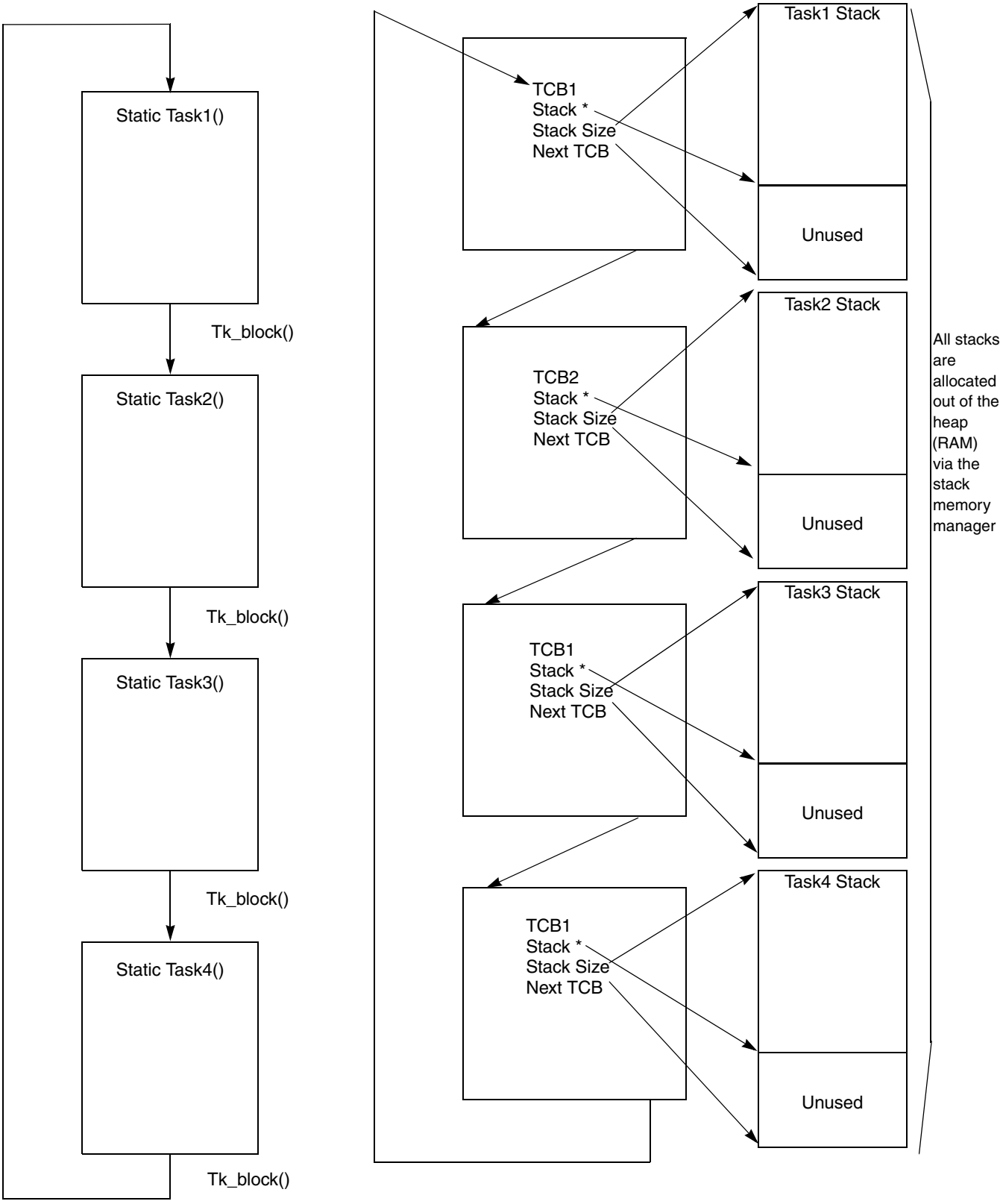


Figure 3. RTOS Structure and Stack Usage

3.1.2 Configuring the RTOS

The RTOS mode is configured in the ipport.h file. There are two defines that set the RTOS mode; they cannot be defined at the same time. If superloop is set to 1, then the RTOS is configured in superloop mode. If the INICHE_TASKS define is set to 1, then the multi-stack mode is enabled. Define one or the other and undefine the other.

3.1.3 RTOS Configuration for Superloop Operation

```
#define SUPERLOOP  1 // EMG 07
// #define INICHE_TASKS    1  // EMG 07 /* InterNiche multitasking system */
```

3.1.4 RTOS Configuration for Multi-Stack Operation

```
// #define SUPERLOOP  1 // EMG 07
#define INICHE_TASKS       1  // EMG 07 /* InterNiche multitasking system */
```

When the RTOS is enabled, there are additional configuration parameters. The additional parameters in osport.h set the stack sizes for each stack. Each stack can be initialized to a unique size. You must consider not only the requirements for the task, but also the requirements of any interrupts when setting a task's stack size. Since any task can be running when an interrupt occurs, the context for the interrupt is stored in the current running task's stack. This requires more RAM when using the RTOS. Since there must be enough room for any interrupts context in all the task's stacks, each stack must be larger than the task requires. This results in wasted RAM.

When the a new task is created, the RTOS allocates RAM from the heap for the new RTOS stack. It then fills this stack with guardwords (defined in task.h). After the stack begins executing, the task naturally writes over the guardwords as it uses the stack. If the task overruns the stack, the RTOS can detect this. This is because the last guardword is overwritten and can throw a fault.

The RTOS needs a global variable cticks (declared in main.c) to be incremented periodically for sleep timing. This is done using a ColdFire® timer. The timer interrupts (timer_isr) are called at a rate initialized by the function clock_init() in main.c.

```
/* task stack sizes */
#define  NET_STACK_SIZE    4096
#define  APP_STACK_SIZE    4096     /* default for applications */
#define  CLOCK_STACK_SIZE  2048

#define  IO_STACK_SIZE     2048
#define  WEB_STACK_SIZE    APP_STACK_SIZE
#define  FTP_STACK_SIZE    APP_STACK_SIZE
#define  PING_STACK_SIZE   2048
#define  TN_STACK_SIZE     APP_STACK_SIZE
#define  IKE_STACK_SIZE    APP_STACK_SIZE
```

3.2 RTOS API

There is no API when using the superloop. Each task returns when it is ready to give up control. With the RTOS enabled, there are more options. Each task has two possible states, sleeping or running. A task can

be asleep if it is time-based or if waiting for an event. When a task is sleeping based on time, it changes to the run state after the timeout. When a task is sleeping on an event, it cannot be run until the event occurs. Events are essentially flags.

When a task is in the sleeping state, it is skipped by the round-robin scheduler. The task cannot run again until the round-robin scheduler comes back to it. If a task is runnable, it is executed when the round-robin scheduler comes to it. The state of the next task is tested by the tk_block() function.

The RTOS does not need interrupts to operate (it is not pre-emptive). A timer must increment the variable cticks (declared globally in main.c) periodically. This global variable is referenced by the scheduler in the tk_block() function to determine how long a task has been sleeping. When a task goes to sleep, the value of cticks + sleep time is stored in the task's TCB. When tk_block() is called, it checks the next task's TCB to see if cticks has passed the value stored in the TCB.

3.2.1 Example: cticks = 100

Sleep (65) sets current_TCB->tk_waketick = cticks + 65 = 165

The current task is also put into a non-running state. cticks is incremented periodically by a timer. The next time tk_block() is executed, it checks if cticks > TCB->tk_waketick. If so, then it sets the task to runnable and runs it.

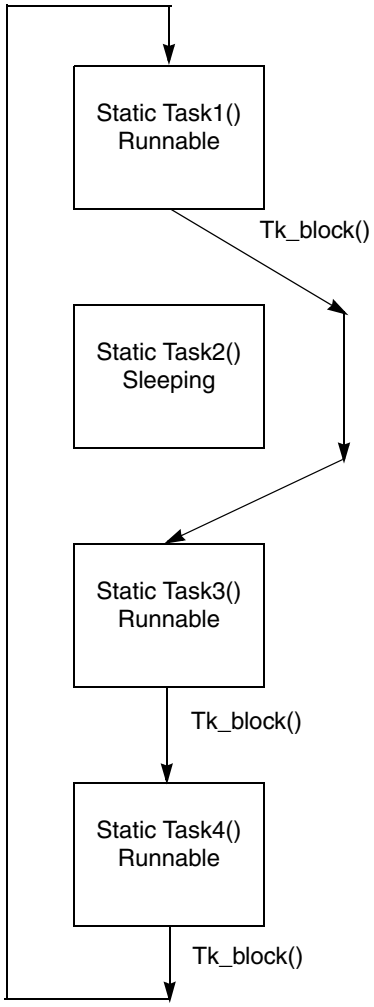


Figure 4. Scheduler Coming to a Sleeping Task

3.2.2 RTOS Functions

The RTOS API has nine functions, listed below:

```
task     * tk_init(stack_t * base, int st_size) // Init the RTOS
```

- Initializes the RTOS and creates the first task. The base address and size of the first task's stack is passed to tk_init() as parameters.
- The first task's default name is "Main".
- The first task's stack is actually on the system stack.

```
task    * tk_new   (task * prev_tk /* predecessor to the new task */

      int    (*start)(int) /* Where the new task starts execution. */

      int     stksiz /* The size in bytes of the stack of the new task. */

      char   * name / * The task's name as a string */
```

```
    int     arg ) /* argument to the task */
```

The tk_new() function creates a new task. The new TCB is inserted after the TCB pointed to by prev_tk. This is as close as it gets to being able to setting a tasks priority. A user can insert a new task anywhere in the TCB ring using this parameter. The TK_NEWTASK wrapper macro (defined in osport.h) calls tk_new() with the prev_tk parameter set to the TCB for the current running task. When this macro is used, the new task is inserted directly after the current running task.

The start parameter is a pointer to the new task's function. The stksiz parameter defines the size of the task's stack. The stack for the new task will be allocated from the heap. The name parameters give the new task a name. The arg parameter passes the new task an integer argument at startup.

```
void    tk_block(void) // switch to next runnable task
```

The tk_block() function first verifies that the current task's stack is not corrupt by verifying the guardword. The tk_block() function then walks the TCB linked list, looking for the next task that is ready to run. The tk_block() function forces a task switch by calling the tk_switch assembly language function in tk_util.s. Interrupts are disabled during task switching.

```
void    tk_exit(void) // kill & delete current task
```

The tk_exit() function sets the current task to be deleted after the next switch. It then forces a task switch. The task is deleted during the task switch.

```
void    tk_kill(task * tk_to_die) // mark any task for death
```

tk_kill() sets the task pointed to by the tk_to_die parameter for death. If the task to die is the current task, it dies as soon as it blocks.

If the task to die is not the current task, it is killed immediately.

```
void    tk_wake(task * tk) // mark a task to run
```

tk_wake() forces the task tk to the run state. The tk task does not run immediately; instead, it is put into a run state so that it runs the next time the scheduler comes around.

```
void    tk_sleep(long ticks) // sleep for number of ticks
```

tk_sleep() puts the current task to sleep for ticks number of cticks.

current_TCB-> tk_waketick = cticks + ticks;

current_TCB->tk_flags &= ~TF_AWAKE; // clear wake flag

tk_sleep() calls tk_block(), causing an immediate task switch.

```
void    tk_ev_block(void * event) // block until event occurs
```

tk_ev_block() puts the current task into a non-running state (sleep) until an event occurs. The event is a 32-bit value.

```
void    tk_ev_wake(void * event) // wake tasks waiting for event
```

Walk through the TCBs, comparing the event parameter to the event entry in the TCB structure. If the TCB's event equals the parameter event then the task is marked as runnable.

Events provide a communication mechanism between tasks. The stack uses events to support blocking sockets. When a socket is waiting for data, it blocks until the data arrives. The stack puts the calling task

to sleep on a unique event (the address of the socket). When data arrives on that socket from the network, the stack wakes the blocking task by sending the event.

3.3 Creating a Task

```
TK_OBJECT(to_keyboard); in tk_ntask.h task *to_keyboard
TK_ENTRY(tk_keyboard); in tk_ntask.h int   tk_keyboard( int parm )

e = TK_NEWTASK(&keyboardtask);  in osporttk.c
  // function adds keyboard task descriptor to tcb list

struct inet_taskinfo keyboardtask =   {
                          &to_keyboard,  // Returns pointer to TCB"console"    // Task name
                          tk_keyboard,     //  Pointer to task function

                          NET_PRIORITY   -  1,  // NOT USED
                          IO_STACK_SIZE, // Stack size
                                   };

TK_ENTRY(tk_keyboard) in tk_ntask.h int tk_keyboard( int parm )
{
   for (;;)
   {
      TK_SLEEP(1);   /* make keyboard yield some time */
      kbdio(); /* let Iniche menu routines poll for char */
      keyboard_wakes++; /* count wakeups */

      if (net_system_exit)
         break;
   }
   TK_RETURN_OK();
}

#define  TK_OBJECT(name)      task *name
```

This macro declares a pointer to a task structure (TCB).

```
#define  TK_ENTRY(name)       int   name(int parm)
```

This macro declares a task function.

```
TK_NEWTASK(struct inet_taskinfo * nettask)
```

Translates the inet_taskinfo into a call to tk_new(). A pointer to the newly created TCB is returned in the pointer pointed to by tk_ptr.

```
new_task = tk_new(tk_cur, nettask->entry,  nettask->stacksize, nettask->name, 0);
*nettask->tk_ptr = new_task;
```

3.4 Menu System and Serial Driver

A simple serial console and menu system is provided for your convenience. The console uses the iuart.c serial driver. The menu system is dynamic and user configurable. The menu subsystem runs as a task. It is implemented in the files menus.c, menulib.c, and nrmenus.c. User applications can insert menu items at run-time using a menu system API. The API is contained in the file menus.c. The file nrmenus.c contains

initial menu items, with the functions associated to each menu item in the menulib.c and nrmenus.c modules.

The install_menu() function inserts a new menu item into the menu array. Each item includes a short description string and a pointer to a function that is called when the command is entered. The menu array is a global structure declared in nrmenus.c.

```
struct menu_op *  menus[] =      /* array of ptrs to menu groups */

// Fill out structure for EMG FFS DIRectory menu command
struct menu_op emg_ffs_dir_menu[] =
{
//          commandfunction         description string
         "EMG HTTP",  stooges,      "EMG HTTP menu",
         "dir",             emg_ffs_dir,    "Dir of EMG FFS",           "flash_erase",
flash_erase,      "Erase the dynamic FLASH area",
         "var",                     emg_http_var,     "Dynamic HTML variable",
         "http",                    emg_http_sessions, "Dump HTTP sessions array",
         NULL,
};

// Install Menu item 'DIR' for EMG FFS
if(install_menu( emg_ffs_dir_menu ) )
printf("\nCould not install DIR menu item for EMG FFS" );
```

The serial driver can be used in an interrupt or polling mode. The driver is configured for polling mode by setting POLLED_UART to 1 in iuart.c. If POLLED_UART is not defined, the driver is in interrupt mode. The baudrate for the driver is set with the UART0_SPEED defined in the iuart.c module. The default baudrate is 115200.

The driver uses both a TX and RX buffer. The buffers are declared out of global RAM (not allocated out of the heap). The defines UART_RXBUFSIZE and UART_TXBUFSIZE set the size of the RX and TX serial buffers, respectively. The RX buffer size should be large enough to hold a complete command line (32 bytes has worked very well). The TX buffer size is a trade-off between performance and RAM. If the TX buffer is too small, application will wait longer for free space in the buffer when sending data to the terminal. If the TX buffer is too big, RAM is being wasted.

3.5 Sample Serial Console Menu Output

INET> help EMG

SNMP Station: EMG HTTP commands:

- dir   - Dir of EMG FFS
- flash_erase - Erase the dynamic FLASH area
- var   - Dynamic HTML variable
- http   - Dump HTTP sessions array
- INET> http
- HTTP sessions array Dump

STATE   VALID   KEEP_ALIVE   FILE_POINTER   SOCKET

```
Wait for header     Not Valid     0     0x0     0x0
Wait for header     Not Valid     0     0x0     0x0
Wait for header     Not Valid     0     0x0     0x0
Wait for header     Not Valid     0     0x0     0x0
INET>
```

3.6 Sample User Defined Menu Options

The INET> prompt is the default prompt. It can be overridden in the allports.c module.

```
INET> tkstats
tasking status:task wakeups: D
name  state          stack   used    wakes
console              running      2048    536     1216676
EMG HTTP server      ready        2048    192     51859563
clock tick           sleeping     2048    104     42047
Main                 blocked      4096    392     0
INET>
```

Above are the tasks for a standard system using the HTTP server. The “Main” task is the first task created by the system using the tk_init() call. The blocked task waits on an event. The sleeping task waits for cticks to be greater then the sleep timer. The ready task is ready to run as soon as the round robin scheduler gets to it. Of course, the console task must also be running to process the command.

The stack column shows the size of the stack defined for each stack. The used column shows the watermark created by the guardword. The tkstats function simply walks the stack from the beginning looking for the guardword.

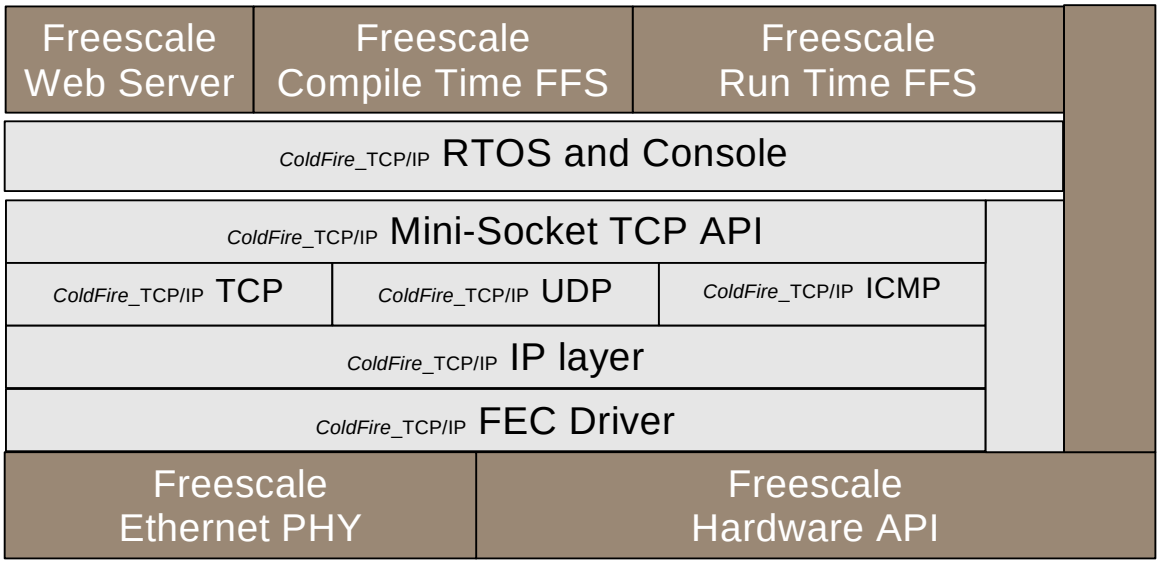


Figure 5. TCP/IP Stack

3.7 TCP/IP Stack Overview

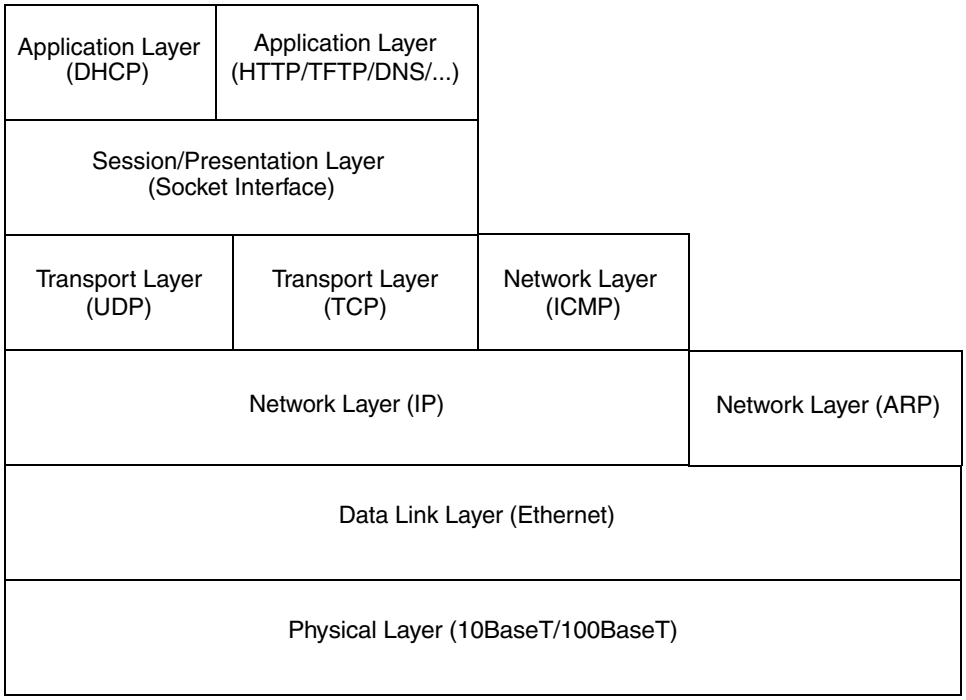


Figure 6. TCP/IP Stack Overview

The TCP/IP stack implements the protocols described in the following RFC's (refer to http://www.rfc-editor.org/rfcxx00.html for details):

- RFC791: Internet protocol (IP)
- RFC792: Internet Control Message protocol (ICMP)
- RFC768: User Datagram Protocol (UDP)
- RFC793: Transmission Control Protocol (TCP)
- RFC826: Ethernet Address Resolution Protocol (ARP)
- RFC1035: Domain Name Server (DNS)
- RFC2131: Dynamic Host Configuration Protocol
- RFC2132: DHCP options

The Session/Presentation layer is a mini-socket interface similar to the BSD socket interface. The stack has been optimized for embedded application using zero-copy functionality for minimum RAM usage.

3.8 Configuring the TCP/IP Stack

The TCP/IP stack is configured from the ipport.h header file. Option macros to trade off features for size. Do not enable options for modules you don't have or your link will receive unresolved externals.

```
*/
#define INCLUDE_ARP             1   // use Ethernet ARP
#define FULL_ICMP               1   // use all ICMP || ping only
#define OMIT_IPV4               1   // not IPV4, use with MINI_IP
#define MINI_IP                 1   // Use Nichelite mini-IP layer
#define MINI_TCP                1   // Use Nichelite mini-TCP layer
#define MINI_PING               1   // Build Light Weight Ping App for Niche Lite
#define BSDISH_RECV             1   // Include a BSD recv()-like routine with mini_tcp
#define BSDISH_SEND             1   // Include a BSD send()-like routine with mini_tcp
#define NB_CONNECT              1   // support Non-Blocking connects (TCP, PPP, et al)
#define MUTE_WARNS              1   // gen extra code to suppress compiler warnings
#define IN_MENUS                1   // support for InterNiche menu system
#define NET_STATS               1   // include statistics printfs
#define QUEUE_CHECKING            1   // include code to check critical queues
#define INICHE_TASKS            1   // InterNiche multitasking system
#define MEM_BLOCKS              1   // list memory heap stats
// EMG #define TFTP_CLIENT          1   // include TFTP client code
// EMG #define TFTP_SERVER          1   // include TFTP server code
// EMG #define DNS_CLIENT           1   // include DNS client code
#define INICHE_TIMERS           1   // Provide Interval timers

// EMG - To enable DHCP, uncomment the line below
#define DHCP_CLIENT             1   // include DHCP client code

// EMG #define INCLUDE_NVPARMS    1   // non-volatile (NV) parameters logic
#define NPDEBUG                 1   // turn on debugging dprintf()s
// EMG #define VFS_FILES            1   // include Virtual File System
// EMG #define USE_MEMDEV           1   // Psuedo VFS files mem and null
#define NATIVE_PRINTF           1   // use target build environment's printf function
#define NATIVE_SPRINTF          1   // use target build environment's printf function
#define PRINTF_STDARG           1   // build ...printf() using stdarg.h
```

```
#define TK_STDIN_DEVICE          1   // Include stdin (uart) console code
#define BLOCKING_APPS            1   // applications block rather than poll
#define INCLUDE_TCP               1   // this link will include NetPort TCP w/MIB

/**** end of option list ***/
```

As stated above, static applications use less RAM if they are not configured for INICH_TASKS and use superloop instead. To do this, undefine INICHE_TASKS and set superloop to 1.

3.9 Setting the MAC and IP Addresses

To enable the DHCP client, set DHCP_CLIENT to 1. If DHCP_CLIENT is not defined, the stack uses the IP address set in the netstatic[] array declared in m_ipnet.c. The netstatic array must be set at zero for the DHCP client to work correctly. If the netstatic IP address is anything other than zero, the DHCP client tries to renew its IP address instead of obtaining a new one.

When DHCP is enabled, the TCP/IP stack cannot finish initializing until after the DHCP transaction is complete. The function netmain_init() in the module allports.c calls the function dhc_setup() in dhcsetup.c. dhc_setup() runs the DHCP protocol which will contact the DHCP server to acquire an IP address and other network related data.

The IP address (DHCP disabled) and MAC address are set using the netstatic[] and mac_addr_fec[] arrays. The netstatic[] array is declared in m_ipnet.c, and the mac_addr_fec[] array is declared in ifec.c.

```
// to set the IP address manually to 192.168.1.99
netstatic[0].n_ipaddr = (0xC0A80163);
netstatic[0].n_defgw  = (0xC0A80101);
netstatic[0].snmask   = (0xffffff00);

// to set the MAC address to 0x00badbad0102
tmp = 0x00badbad;
mac_addr_fec[0] = (u_char)(tmp >> 24);
mac_addr_fec[1] = (u_char)(tmp >> 16);
mac_addr_fec[2] = (u_char)(tmp >> 8);
mac_addr_fec[3] = (u_char)(tmp & 0xff);

tmp = 0x01020304;
mac_addr_fec[4] = (u_char)(tmp >> 24);
mac_addr_fec[5] = (u_char)(tmp >> 16);
```

These structures must be configured before the stack is initialized with a call to netmain_init() in the module allports.c

3.10 DHCP Client

The Dynamic Host Configuration Protocol acquires network parameters at runtime. The protocol uses the UDP layer of the stack. The stack must be initialized with a call to ip_startup() before the DHCP client can be called.

The DHCP protocol is defined in RFC2131 and RFC2132. The stack runs a DHCP client which searches for a DHCP server (this is referred to as discovery). Packets are transferred using the UDP layer and BOOTP ports (67 and 68). Since the IP stack does not have an IP address yet, it discovers using broadcast addresses. Included in the discovery packet is a unique transaction ID (xid). A listening DHCP server

sends an offer message containing the xid sent by the client and the suggested network parameters, again using broadcast addressing. Also encoded in the offer is a unique server ID. The client uses this server ID when sending a request packet back to the server indicating that it accepts the network parameters that were offered. Finally, the server ACKS the client using it's new IP address.

RFC2132 specifies various options that can be requested by the DHCP client. These options can also report information to the DHCP server. The options supported by the DHCP client are listed in the dhcpclnt.h module. Two reporting options of special interest are 12 and 15 (DHOP_NAME and DHOP_DOMAIN). These two options are passed to Domain Name Servers (DNS) by most DHCP servers. The DHCP client is contained in the modules dhcpclnt.c, dhcpclnt.h, and dhcsetup.c.

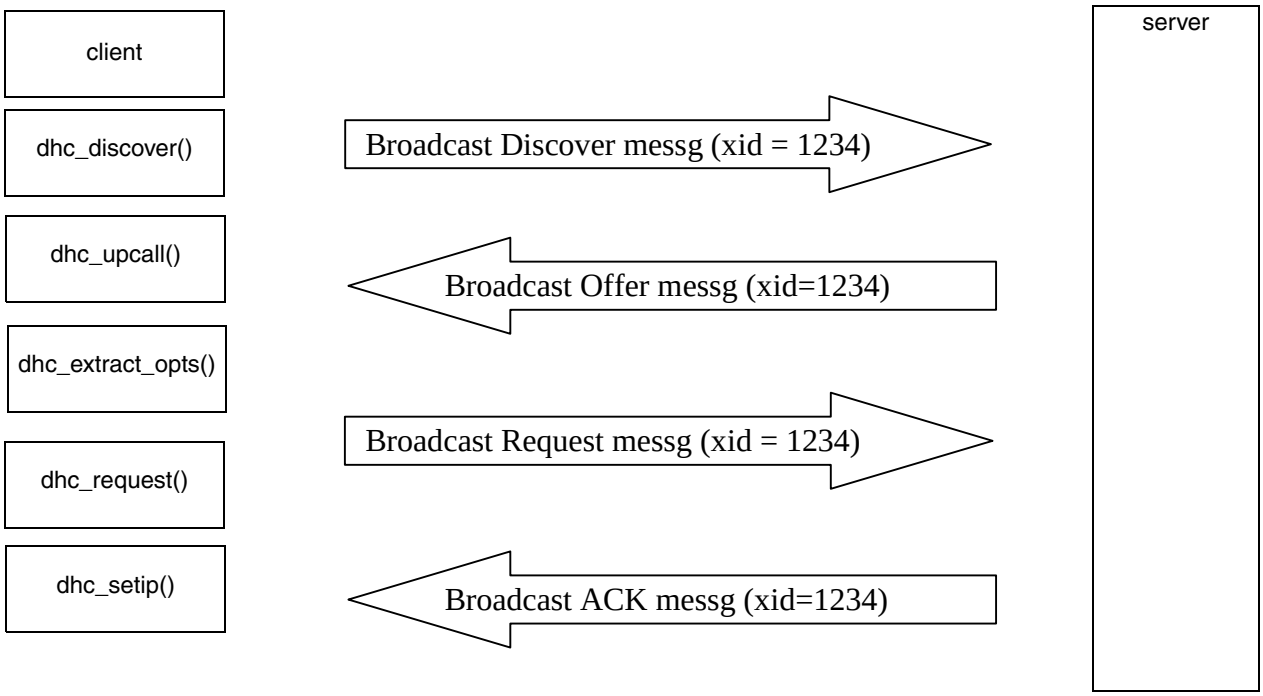


Figure 7. DHCP Client

The netstatic[] structure must be cleared to zero before calling dhc_setup() to start the DHCP transactions. If not, the DHCP client attempts to renew whatever IP address is in the netstatic[] structure. This is a valid process only if the IP address in the netstatic[] structure was originally provided by the DHCP server.

3.11 DNS Client

The DNS client communicates with the DNS (Domain Name Server). The DNS system translates domain names into IP addresses. The DNS protocol is described in RFC1035. DNS can use UDP or TCP, with port 53. The DNS protocol is stateless. All the information is contained in a single message. This message is fully documented in RFC1035. Figure 8 shows the DNS message.

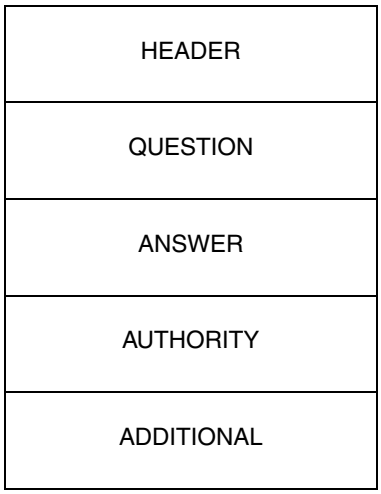


Figure 8. DNS Message

- Question: The question for the name server
- Answer: RR's answering the question
- Authority: RR's pointing toward authority
- Additional: RR's holding additional information

The DNS client is enabled by setting the DNS_CLIENT macro to 1 in the ipport.h file. The DNS client is maintained by calling the dns_check() function every second. This keeps the DNS cache up to date. The DNS client must be initialized by filling the dns_server[] array with the IP addresses of DNS name servers. The dns_servers[] array is declared globally in dnsclnt.c. Unused entries should be filled with zeros.

```
ip_addr dns_servers[MAXDNSSERVERS]
```

Calls an API function to use the DNS client. When first requested, a name translation must be called using either the dns_query(), dns_query_type(), or gethostbyname() functions. Each of these functions inserts a name and returned IP address to a cache. After the query is performed once, the dns_lookup() function gets the information from the cache.

3.12 Stack RAM Usage

The TCP/IP stack uses RAM for packet storage. Packets are stored in buffers managed by a dedicated buffer queue, not the heap manager. The module pktalloc.c contains the packet buffer memory manager. There are two categories of packet buffers, based on the size of the buffer. bigbufsiz is declared in pktalloc.c and determines the size of the big buffers. lilbufsiz is declared in pktalloc.c and determines the size of the small buffers.

Ethernet-receive operations use only the bigbufs, while transmit operations use either bigbufs or lilbufs depending on the size of the packet. The receive operation must use bigbufs only because the size of the packet received is not known until the whole packet is received. Bigbufsiz is set to be larger then any received packet.

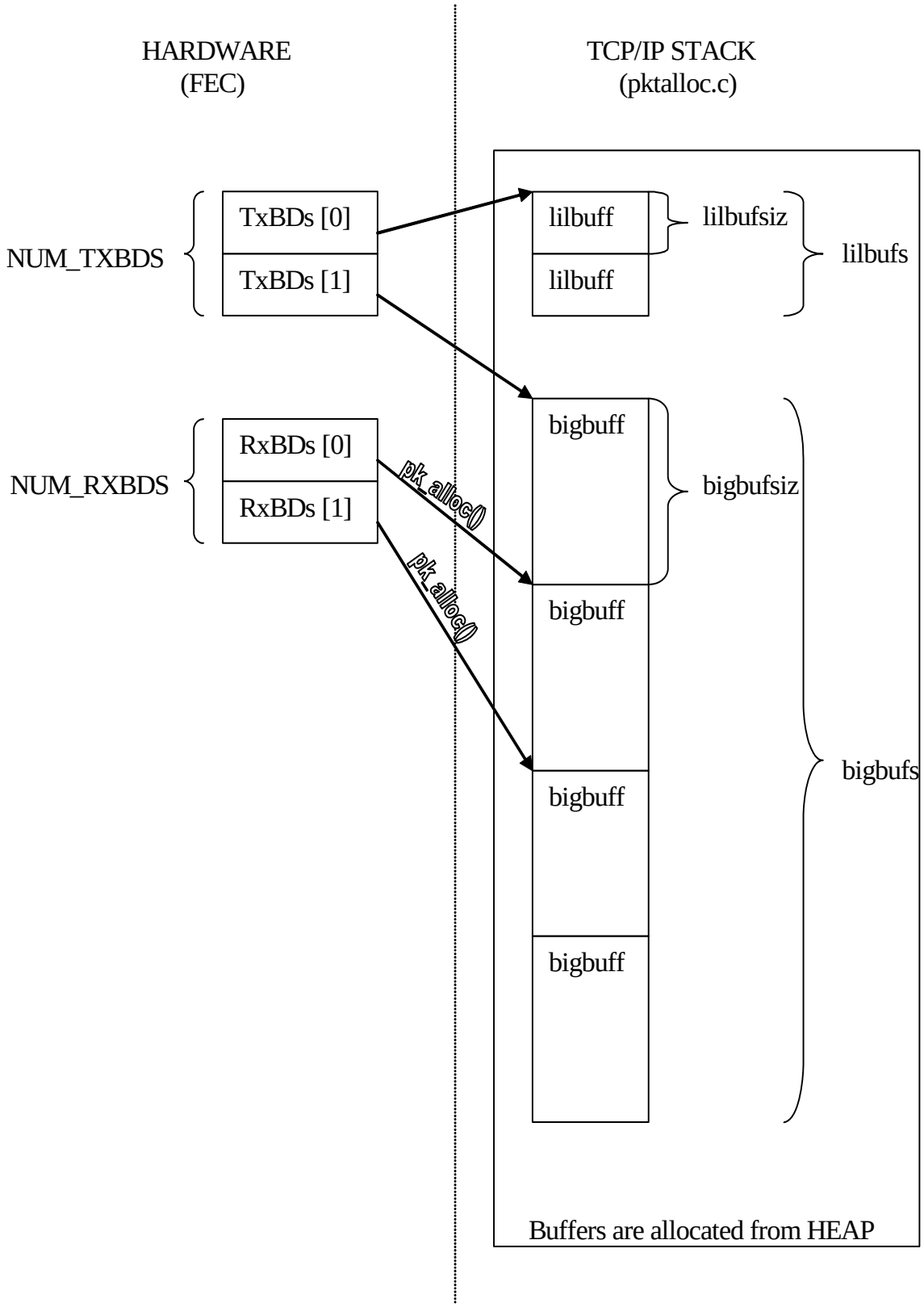


Figure 9. Stack RAM Usage

When the stack wants to send a packet, it calls pk_alloc(length) with a length equal to the desired packet length. Pk_alloc() uses either a bigbuf or lilbuf depending on the desired packet size. This is done for RAM efficiency. For TCP applications, make sure lilbufsiz is greater then the size of a ACK packet (60 bytes with ETHER header).

RX buffers must be large enough to accept a full size ethernet packet. The macro MAX_ETH_PKT defined in fecport.h sets the maximum packet size that can be received or transmitted by the Fast Ethernet

Controller (FEC). Bigbufsiz must be larger then MAX_ETH_PKT. The FEC requires that the packet buffer be on a 16 byte boundary. Add 16 bytes to the buffsize to accommodate alignment.

- bigbufsiz >= MAX_ETH_PKT + 16
- lilbufsiz should be greater then TCP ACK packet size (60)

The number of buffers is a trade-off between performance and available RAM. The stack allocates packet buffer RAM from its heap during init. The packets buffer RAM is never returned to the heap. They are managed by a separate independent memory manager (pktalloc.c). When determining the number of buffers, you must consider network performance requirements and traffic. Even if the embedded application does not require heavy ethernet traffic, if there is heavy traffic on its network it will need more buffers. Although the FEC filters ethernet addresses, broadcast addresses from ARP requests are passed to the stack using packet buffers. With a small number of packet buffers, broadcast packets can have a drastic effect on stack performance. It is important that any changes to the number of bigbufs be tested in a network environment similar to the environment the final device will be used in.

TCPTV_MSL defines the amount of time a TCP connection will wait in the CLOSE-WAIT state. A socket does not close immediately. It waits TCPTV_MSL * 2 seconds before actually closing and releasing the packet buffers. In an environment where the connection is frequently opened and closed (e.g. a webserver), waiting too long to free up a packet buffer from a previously closed connection can result in all the packet buffers being locked up waiting for TCPTV_MSL timeouts.

TCP_MSS or TCP Maximum Segment Size sets the max number of data bytes a TCP segment can hold. This value must be smaller then bigbufsiz. In fact, the MSS must be less then (bigbufsiz – TCP header – IP header – Ethernet header). When sending large amounts of data, the higher the TCP_MSS the better (within the limits mentioned above). The total data sent is broken up into (total data size)/TCP_MSS TCP segments. A greater the number of segments leads to more overhead to ACK the segments, causing worsened performance.

3.13 Tested TCP/IP Stack Parameters

```
MAX_ETH_PKT (in fecport.h) = 1520;
bigbufsiz (main.c) = 1542 // could be reduced to 1536
lilbufsiz (main.c) = 200 // could be reduced to 76 depending on app
                          // if app is sending small packets > 60 then
                         // increase to small packet size + 16
bigbufs (NUMBIGBUFS ipport.h) =  8 // 8 * 1542 = 12336
lilbufs (NUMLILBUFS ipport.h) = 6 // 6 * 200 = 1200
TCP_MSS = 1456           // Allows room for TCP options
TCPTV_MSL = 1            // Fast close of sockets
```

Using the numbers above requires 12336 + 1200 bytes of heap space (RAM) just for the packet buffers.

3.14 Reduced RAM TCP/IP Stack Parameters

```
MAX_ETH_PKT (in fecport.h) = 1520
                                // The MTU size is maintained at the
                                // internet standard 1520
                                // This guarantees compatibility
bigbufsiz (main.c) = 1542;      // could be reduced to 1536
                                // Based on MAX_ETH_PKT
lilbufsiz (main.c) = 100;       // Reduced assuming an app that sends
```

```
                                    //  about 20 bytes / packet
bigbufs (NUMBIGBUFS ipport.h) =  4 // 4 * 1542 = 6168
lilbufs (NUMLILBUFS ipport.h) = 4 // 4 * 100 = 400
TCP_MSS = 1456                      // Allows room for TCP options
TCPTV_MSL = 1                       // Fast close of sockets
```

Using the above parameters reduces the stack heaps requirements for packet buffers to 6168 + 400 = 6568 bytes or 6.4 K

3.15 TCP/UDP/IP Stack API

The mini-sockets API is designed to be as close as possible to the BSD Sockets API while still allowing a small footprint. The primary differences are that passive connections are accomplished with a single call, m_listen(), rather than the BSD bind()-listen()-accept() sequence. The BSD select() call is also replaced with a callback mechanism.

BSD = Berkeley Software DistributionTCP/UDP/IP Stack IP

Mini-Sockets	BSD Sockets
m_socket()	socket()
m_connect()	connect()
m_recv() and/or m_send() - or - tcp_send() and/or tcp_recv() - (zero-copy I/O)	recv() and/or send()
m_close()	close();

For server applications:

Mini-Sockets	BSD Sockets
(n/a - merged with listen)	socket()
(n/a - merged with listen)	bind()
m_listen()	listen()
(n/a - handled via callback)	accept()
m_recv() and/or m_send() - or - tcp_send() and/or tcp_recv() - (zero-copy I/O)	recv() and/or send()
m_close()	close();

Figure 10. TCP/UDP/IP Stack IP

`char *ip_startup()`

- Initializes the TCP/IP stack. It returns null if the init is OK. If not, it returns an error string.

`int input_ippkt(PACKET pkt, int length)`

- Called by the ISR after a packet has been received from the hardware. This function inserts a received ethernet packet into the stack. It always returns 0. The packet typdef is a pointer to a netbuf structure.

```
struct netbuf
{
                                    struct netbuf * next;   // queue link
```

```
                                    char   * nb_buff;    // beginning of raw buffer
                                    unsigned nb_blen;    // length of raw buffer
                                    char   * nb_prot;    // beginning of protocol/data
                                    unsigned nb_plen;    // length of protocol/data
                                    long     nb_tstamp;  // packet timestamp
                                    struct net *   net;  // the interface it came in on, 0-n
                                    ip_addr  fhost;      // IP address associated with packet
                  unsigned short type; // IP==0800 filled in by recever(rx)
                                                or net layer.(tx)
                                    unsigned inuse;      // use count, for cloning buffer
                                    unsigned flags;      // bitmask of the PKF_ defines
                                    char *   m_data;        // pointer to TCP data in nb_buff
                                    unsigned m_len;         // length of m_data
                                    struct netbuf * m_next; // sockbuf que link
                                    struct ip_socopts *soxopts; // socket options */
};
```

`void packet_check( )`

- TCP/IP stack state machine. Must be called periodically by a task or from the superloop.

`void dhc_setup()`

- Called after ip_startup() to kick off the DHCP client. The stack cannot be used until the DHCP transaction is complete.

`M_SOCK m_socket()`

- Allocates a socket structure. The socket defaults to blocking. Returns a MSOCK structure if OK, NULL if error.

`int m_connect(M_SOCK so, struct sockaddr_in * sin, M_CALLBACK(name))`

- Starts the connection process to a server. The m_connect() function attempts to connect to the IP address and the port specified in the sockaddr_in structure. If the socket is flagged as blocking, m_connect() does not return until a timeout defined by TCPTV_KEEP_INIT (in mtcp.h) which defaults to 75 seconds. If the socket is flagged as non-blocking (via the m_ioctl() function) then m_connect() returns EINPROGRESS. When the socket is flagged as non-blocking, the M_CALLBACK parameter signals a completed connection by calling the M_CALLBACK function.

The m_connect() function returns the error codes specified in the file msock.h (see below). The MSOCK typedef is a pointer to an msocket structure.

```
struct msocket
{
        struct msocket * next;       // queue link
        unshort  lport;             // IP/port tupple describing connection, local port
        unshort  fport;             // far side's port
        ip_addr  lhost;             // local IP address
        ip_addr  fhost;             // far side's IP address
        struct tcpcb * tp;          // back pointer to tcpcb
        struct m_sockbuf sendq;       // packets queued for send, including unacked */
        struct m_sockbuf rcvdq;       // packets received but undelivered to app
        struct m_sockbuf oosq;        // packets received out of sequence
        int      error;             // last error, from BSD list
        int      state;               // bitmask of SS_ values from sockvar.h
        int      so_options;        // bitmask of SO_ options from socket.h
        int      linger;            // linger time if SO_LINGER is set
        M_CALLBACK(callback);         // socket callback routine
```

```
        NET      ifp;              // iface for packets when connected
        char     t_template[40];     // tcp header template, pointed to // by tp->template
        void *   app_data;         // for use by socket application
};


struct sockaddr_in
{
        short    sin_family;
        u_short  sin_port;
        struct           in_addr  sin_addr;
        char             sin_zero[8];
};
```

The sockaddr_in structure is used extensively by the TCP/IP stack API.

- sin_family must be set to AF_INET.
- sin_port is the 16 bit port number.
- sin_addr is the 32 bit IP address.
- sin_zero[] is not used.

```
M_SOCK m_listen (struct sockaddr_in * sin, M_CALLBACK(name), int * error)
```

- The m_listen() function creates its own socket (by calling m_socket()) and listens for a connect from a remote client. Using the sockaddr_in structure, the listening socket can be assigned a port to listen on, as well as an IP address to listen to. If the IP address is 0, then m_listen() will listen to any IP address. If the server only accepts a connection from a single specified IP address, it sets the IP address in the sin_addr. If the server accepts connections from any IP address (standard operation), it sets sin_addr to 0. The callback function is called when a connection is established.

Sets:

```
socket->fhost = sockaddr_in->sin_addr;   // far side IP
socket->lport = sockaddr_in->sin_port;   // local port
```

```
int m_close(M_SOCK so)
```

- Closes any open connections on the socket, and releases the socket.

```
int m_send(M_SOCK so, char * data, unsigned datalen)
```

- The m_send() function is enabled when the BSDISH_SEND macro is defined. The m_send() function is a “workalike” for the BSD sockets send() function
- Data points to the byte array to send, and datalen is the length of the array
- Returns the number of bytes actually sent on success, or -1 on error

```
int m_recv(M_SOCK so, char * buf, unsigned buflen)
```

- The m_recv() function is enabled when the BSDISH_RECV macro is defined. The m_recv() function is a “workalike” for the BSD sockets recv() function.

The buflen parameter specifies the max number of bytes that the stack can store in the buf array. The buf array is filled with data from the stack. buf will not necessarily be on packet boundaries. The buf array can hold only a portion of a packet or multiple packets. m_recv() and m_send() are streaming functions, in that there are no boundaries or chunks of data.

```
int m_ioctl(M_SOCK so, int option, void * data)
```

- Ioctl is used to configure various socket options.

- The options parameter is used to select the option, and the data parameter is used to select the value.

Options (defined in msock.h):

- SO_NONBLOCK sets socket to non-blocking
- SO_BIO sets socket to blocking
- SO_NBIO sets/clears blocking based

Data!=0 is not blocking

- SO_DEBUG sets socket debug mode

With the DO_TCPTRACE

Macro sets, enables TCP

Debug messages

- SO_LINGER sets the time to wait before

Closing a socket

If set to 0, socket closes

Immediately with a call

To m_close().

3.16 TCP/IP Stack Zero-Copy API

The TCP/IP stack supports a zero-copy API. The primary advantage of this API is speed, but the disadvantage is overhead. When using this API, the application manages packet buffers.

```
PACKET tcp_pktalloc(int datasize)
```

- Allocates a packet buffer by calling pk_alloc(datasize+headersize).
- Headersize is hardcoded to 54 bytes when the MINI_IP macro is defined. pk_alloc() will kick out a request for a packet bigger then bigbufsiz, so datasize must be less then (bigbufsiz – 54).
- The PACKET returned is a pointer to a structure of type netbuf.

```
int tcp_send(M_SOCK so, PACKET pkt)
```

- Sends a packet allocated by tcp_pktalloc(). The application should copy its data to *pkt->m_data and the number of bytes to pkt->m_len. Returns 0 if everything is okay and a error code for failure.
- Error codes can be found in msock.h.
- If a 0 is returned, the stack owns the packet and will free it after sending. If an error message is returned, the application still owns the packet and must return it.

```
PACKET tcp_recv(M_SOCK so)
```

- Returns the next packet the socket receives. Packet data is pointed to by pkt->m_data, with data length in pkt->m_len. The application must free the packet after it is done processing the data.
- Returns pointer to netbuf structure after packet is received, or NULL if no packet received and socket is non-blocking.

```
void tcp_pktfree(PACKET p)
```

- Free’s netbuf pointed to by p.

3.17 UDP/IP Stack API

```
UDPCONN udp_open(ip_addr fhost, insert fsock, unshort lsock,
                            int (*handler)(PACKET, void*),  void *   data)
```

- The udp_open() function inserts a callback entry into the udp demux array. When udpdemux() in m_udp.c is passed a UDP packet from the IP layer, it searches a link list of udp_conn structures (defined in udp.h) looking for a callback function associated with the port number from the incoming packet. When udpdemux() finds the correct callback function, it calls the function passing it the packet from the IP layer. The udp_open() function fills in the udp_conn structure from the input parameters, then inserts the new udp_conn structure into the linked list. Data points to a 32-bit value that will be passed to the callback function when it is called. The IP stack only passes up packets addressed to us (promiscuous mode is not supported).

```
udp_conn->u_lport = lsock// Local port number
udp_conn->u_fport = fsock// Foreign port number
udp_conn->u_fhost = fhost// Foreign IP address
```

- Returns a pointer to a udp_conn structure, or NULL if error.

```
void udp_close(UDPCONN con)
```

- Removes the udp_conn structure from the linked list. udp_close()
- Frees up the udp_conn structure from the heap. Any UDP packet sent to the port identified in the udp_conn structure is dropped by the udpdemux() function.

```
PACKET udp_alloc(int datalen, int optlen)
```

- Allocates a packet buffer of size datalen + 34 + optlen. The packet buffer must be smaller then bigbufsiz.
- Returns a pointer to a netbuf packet on success, and NULL on error.

```
void udp_free(PACKET p)
```

- Free’s netbuf pointed to by p.

```
int udp_send(unshort fport, unshort lport, PACKET p)
```

- Sends the packet pointed to by p from lport to foreign port.
- Copies data to *p->nb_prot and set p->nb_plen = number of bytes.
- Returns 0 on success, or error code.

3.18 TCP/UDP/IP Stack API Return Codes

- ENOBUFS: No packet buffers of requested size available
- ETIMEDOUTTCP: Timeout occurred (No ACK, no connection)
- EISCONN: m_connect() error occurs if socket is already connected.
- EOPNOTSUPP: m_ioctl() error occurs if option is not supported.
- ECONNABORTED: Not used
- EWOULDBLOCK: Indicates that a non-blocking socket would block. This error would occur if a socket is set to non-blocking and a m_recv() function had been called when the stack had no data to feed the m_recv().
- ECONNREFUSED: Connection was refused by the server
- ECONNRESET: Connection reset by the host
- ENOTCONN: An attempt was made to communicate on a socket that was not connected.

- EALREADY: Not used
- EINVAL: Socket passed to API is not valid. It may have been closed.
- EMSGSIZE: Not used
- EPIPE: Error from tcp_send(); host closed the connection.
- EDESTADDRREQ: Not used
- ESHUTDOWN: Socket disconnected
- ENOPROTOOPT: Not used
- EHAVEOO: Not used
- ENOMEM: Error allocating memory for socket or other required structure
- EADDRNOTAVAIL: Muti-cast address not found.
- EADDRINUSE: Muti-cast address in use
- EAFNOSUPPORT: Not used
- EINPROGRESS: Non-blocking error indicating connection in place (m_connect())
- ELOWER: Not used

3.19 DHCP Client API

```
void dhc_setup( void )
```

- Initializes the DHCP client. The client attempts to acquire an IP address for 30 seconds, then fails. The function does not return until an IP address is acquired (DHCP in the BOUND state) or the timer times out. The 30 second timeout is specified in the dhc_setup() function in dhcsetup.c. The timeout is hardcoded in a while loop about ¾ of the way into the function (look for TPS).
  - while (((cticks - dhcp_started) < (30*TPS)) &&
  - (dhc_alldone() == FALSE))

```
int dhc_second(void)
```

- This function is in dhcpclnt.c. It must be called once each second to support the DHCP specification for lease times and IP renews and expirations fully.

3.20 DNS Client API

```
int dns_query(char * name, ip_addr * ip_ptr)
```

- Requests a host name to IP address translation
- The name parameter is the host name string. The ip_ptr will be filled in with the IP address if available.
- Returns 0 on successful translation; otherwise, it returns an error number.

```
int dns_query_type(char * name, char type, struct dns_querys ** dns_ptr)
```

- Requests a specified type of data from the name server.
  - Types: DNS_TYPE_QUERY // Type value for question
    - DNS_TYPE_IPADDR // Type value for IPv4 address
    - DNS_TYPE_AUTHNS // Authoritative name server
    - DNS_TYPE_ALIAS // Alias for queried name

```
void dns_check(void)
```

- Should be called once a second to support DNS timeouts and retries

```
int dns_lookup(ip_addr * ip, char * name)
```

- Looks in DNS cache for name-to-IP address translation.
- If found in cache, returns 0.

```
struct hostent *gethostbyname(char * name)
```

- “Standard” API for name translation. Returns pointer to hostent structure on success, NULL on failure. Hostent is defined in dns.h.

3.20.1 Example UDP Server Pseudo Code

A simple UDP server that will process packets from any IP address or port, sent to my ip address and port 1234.

```
#define EMGDATA           ((void*)0x12345678)
UDPCONN emg_conn = NULL   // Declare a pointer to a UDP connection
                          // structure
emg_conn = udp_open( 0,   // accept UDP packets addressed to us
                          // from any ip address 0,
                          // accept UDP packets FROM any port 1234,
                          // Only accept UDP packets sent to my
                          // port number 1234 emg_upcall,
                          // Callback Function EMGDATA );
                          // 32 bits value sent to emg_upcall
```

Now, any UDP packet addressed to my IP and port number 1234 will be passed to the callback function emg_upcall().

```
int emg_upcall( PACKET pkt, void *data )
{
        int     datalength = pkt->nb_len;// datalength = # of bytes received
        u_char *data;

        data = (u_char *)pkt->nb_prot;    // data[0] = first byte of rx data

        udp_free(pkt);                    // If we return 0,  we must free the packet
        return(0);
//      return(-1);                      // Returning anything but zero will force
                                         // the stack to throw out the packet and Free
                                         // the packet buffer
}
```

3.20.2 Example UDP Client Pseudo Code

A simple UDP client that will send data (‘eric’) from local port 5678 to foreign port 1234, ip address 192.168.1.1.

Since this is a send only app, no need to call udp_open().

```
PACKET   pkt;                            // Declare a pointer to a packet
u_char           *dataout                // Pointer to output data

pkt = udp_alloc( 100,                    // Request to send 100 bytes or less 0 );
                                         // No addition bytes needed if( pkt != NULL )
{
```

```
        dataout = (u_char *)pkt->nb_prot;   // dataout points to first byte in data portion
                                            // of buffer
        dataout[0] = 'e';                   // Copy data to packet buffer
        dataout[1] = 'r';
        dataout[2] = 'i';
        dataout[3] = 'c';

        pkt->nb_plen = 4;                   // Set data length

        udp_send( 1234,                     // Send to port 1234
                  5678,                     // From local port 5678
                     pkt );
}
```

3.20.3 Example TCP Server Pseudo Code

3.20.3.1 Creating a Listening Socket

```
// Init a socket structure with our Port Number
emg_http_sin.sin_addr.s_addr       = (INADDR_ANY);
emg_http_sin.sin_port              = (PORT_NUMBER);
emg_http_server_socket             = m_listen(&emg_http_sin, freescale_http_cmdcb, &e);
```

3.20.3.2 Accepting a Connection

```
switch(code)
{
        // socket open complete
        case M_OPENOK:
                msring_add(&emg_http_msring, so);
                break;
}
```

3.20.3.3 Receiving TCP Data

```
length = m_recv( freescale_http_sessions[session].socket, (char *)buffer, RECV_BUFFER_SIZE );
```

3.20.3.4 Sending TCP Data

```
bytes_sent = m_send( freescale_http_sessions[session].socket, data, length);
```

3.20.3.5 Closing the Socket

```
j = m_close(so);
```

3.20.4 Example TCP Client Server Code

3.20.4.1 Creating a Socket

```
M_SOCK Socket = m_socket();
```

3.20.4.2 Connecting to a Server

```
int m_connect(M_SOCK socket, struct sockaddr_in * sin, M_CALLBACK(name));
// m_connect is blocking until a connection completes.
// If the socket is configured for non-blocking, then the callback function is used to indicate
when the connection is established.
```

3.20.4.3 Receiving TCP Data

```
length = m_recv( freescale_http_sessions[session].socket, (char *)buffer, RECV_BUFFER_SIZE );
```

3.20.4.4 Sending TCP Data

```
bytes_sent = m_send( freescale_http_sessions[session].socket, data, length );
```

3.20.4.5 Closing the Socket

```
j = m_close( so );
```

3.20.5 TCP/IP Stack Packet Flow

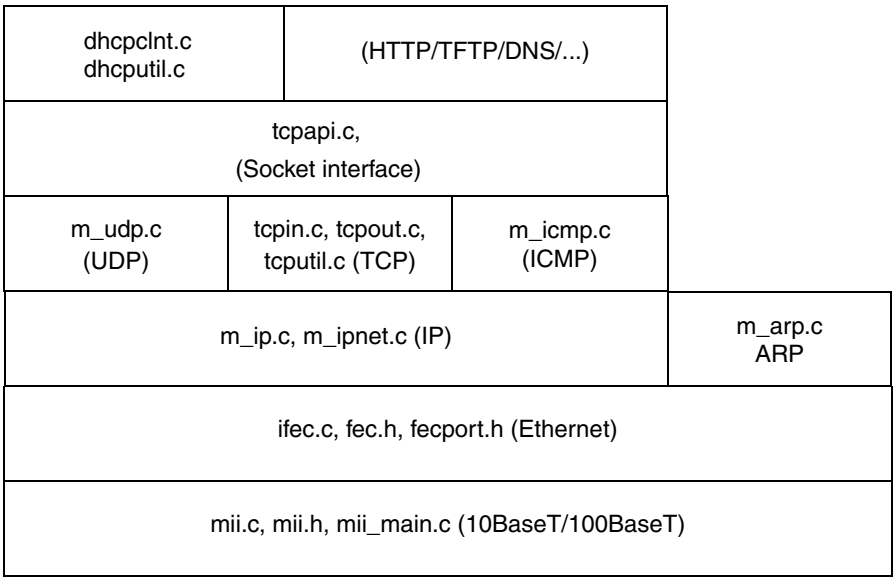


Figure 11. TCP/IP Stack Packet Flow

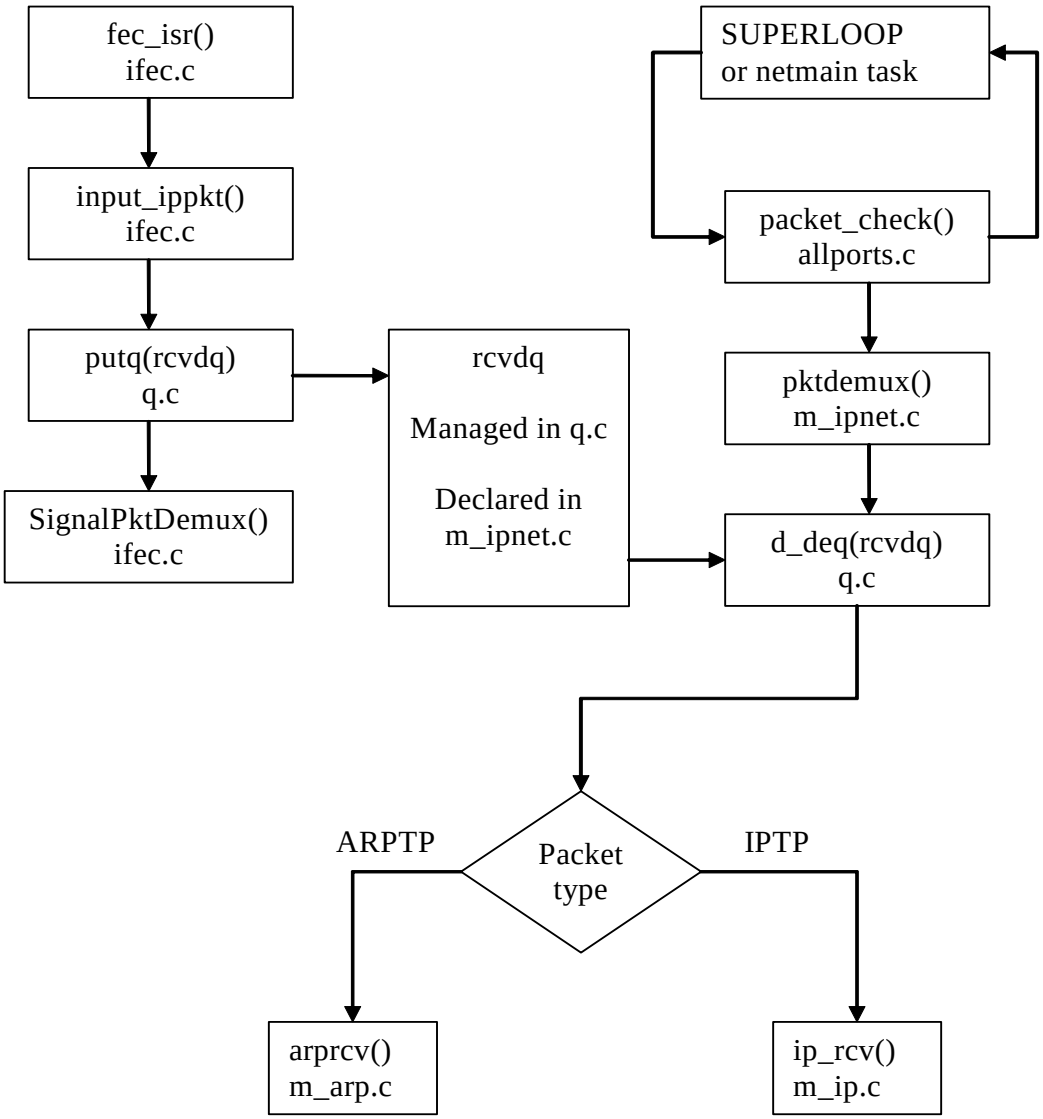
When a packet is received, the FEC ISR passes the packet to the stack by calling input_ippkt() in ifec.c with a pointer to the packet and the length of the packet. The packet is inserted into an input packet queue with a call to putq() in the module q.c. Then the function SignalPktDemux() is called to wakeup the TCP/IP stack.

The function packet_check() in allports.c must be called routinely from either a task or the superloop. This function calls pktdemux() in m_ipnet.c to start the actual processing of the packet. The pktdemux() function looks at the ethernet header to determine the packet type. ARP packets are sent to the arprcv() function for processing, while IP packets are sent to the ip_rcv() function.

If the packet is type ARPTP, the arprcv() function in m_arp.c is called. Address Resolution Protocol, or ARP, is used to match an IP address with an ethernet MAC address. The arprcv() function handles both ARP requests from other devices. ARP replies from other devices. ARP requests from other devices cause the arpReply() function in m_arp.c to be called, which will send out information to the other device. ARP replies from other devices are responses to ARP requests. These replies are stored in the ARP table.

If the packet is type IPTP, then the ip_rcv() function in m_ip.c is called. The ip_rcv() function filters the IP layer of the packet. It first verifies that the packet is the correct type. The free version of the stack only supports IPV4. IPV6 packets are dropped. The checksum for the IP layer of the packet is then tested. If the IP layer checksum is bad, the packet is dropped. Finally, the packet address is checked to see if it is addressed to us. Packets not addressed to us are dropped. After the packet is verified for type, checksum, and IP address, the packets is passed up the stack depending on the protocol field in the IP header.

The ColdFire TCP/IP stack supports three IP protocols: UDP, ICMP, and TCP. User Datagram Packets, or UDP, are passed up to the udpdemux() function. ICMP packets are passed to the icmprcv() function. TCP packets are passed to the tcp_rcv() function.


fec_isr()
ifec.c
input_ippkt()
ifec.c
putq(rcvdq)
q.c
SignalPktDemux()
ifec.c
rcvdq
Managed in q.c
Declared in
m_ipnet.c
SUPERLOOP
or netmain task
packet_check()
allports.c
pktdemux()
m_ipnet.c
d_deq(rcvdq)
q.c
Packet
type
ARPTP
IPTP
arprcv()
m_arp.c
ip_rcv()
m_ip.c

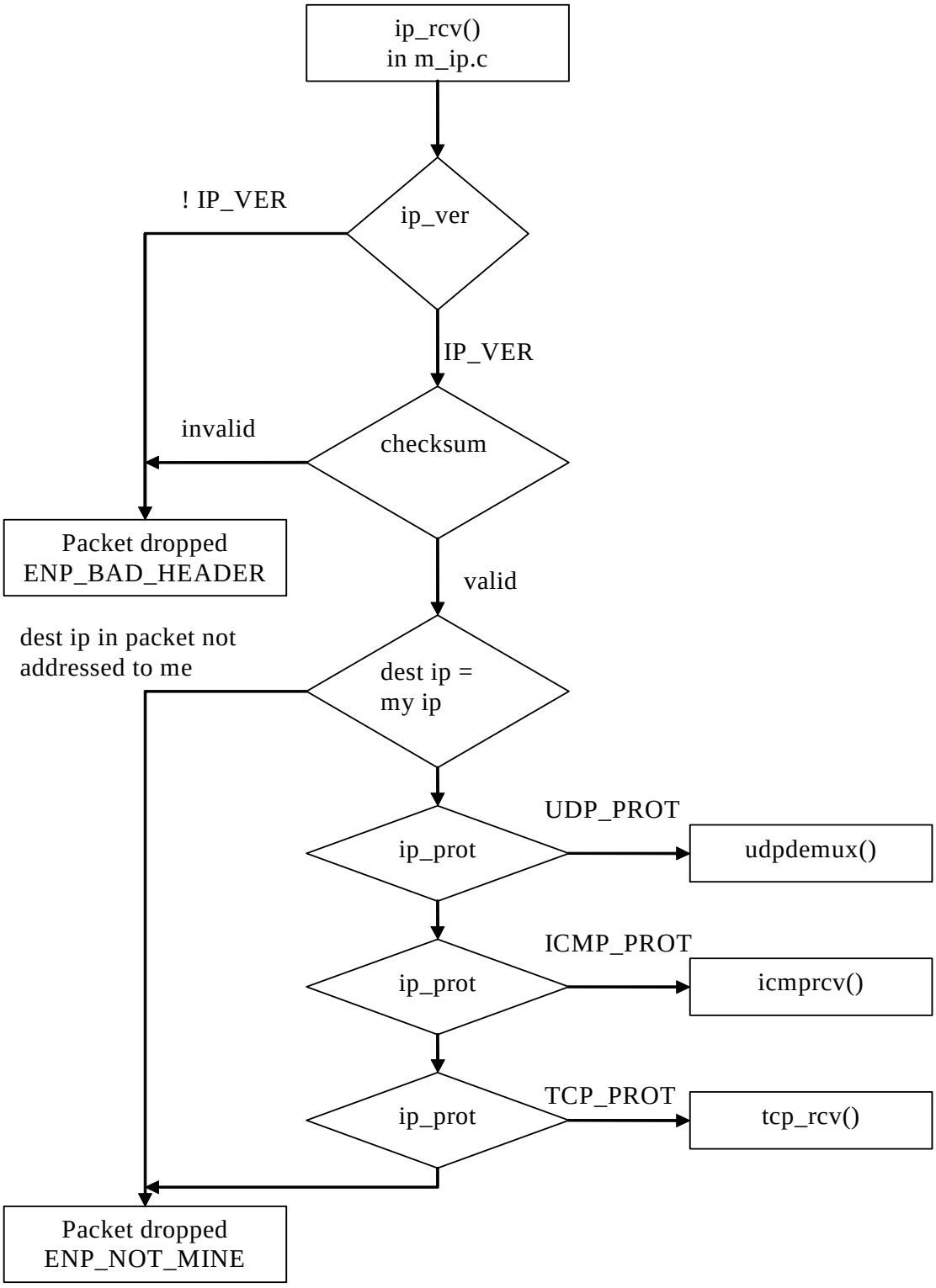


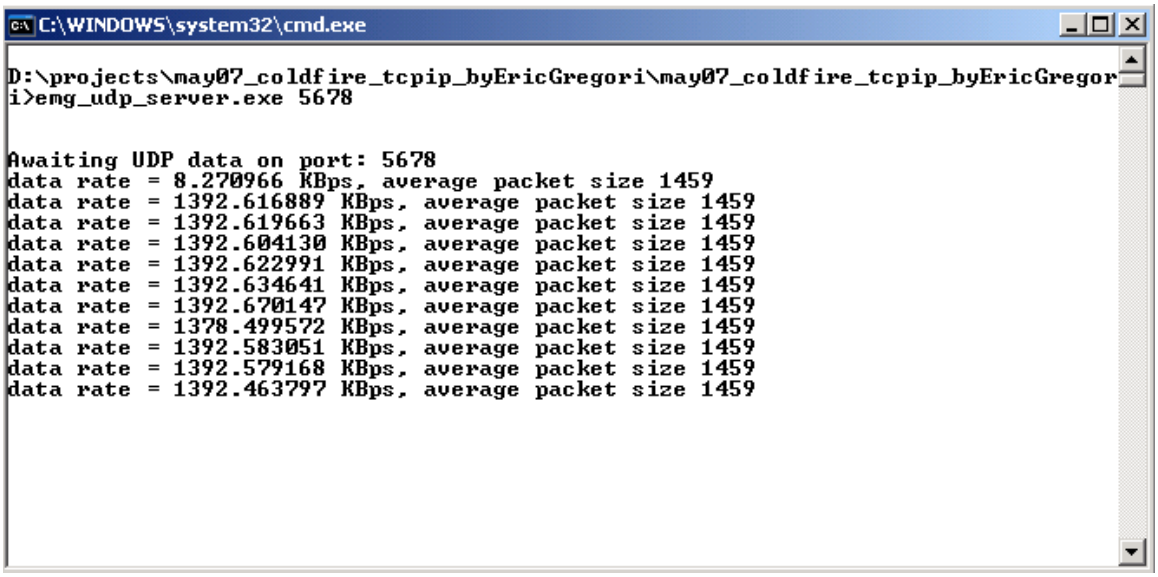
Figure 12. ColdFire TCP/UDP/IP Stack

3.20.6 TCP/UDP/IP Stack Performance

The stack performance was measured using custom PC applications included in the project zip file. The targets mentioned are part of the CodeWarrior project associated with this application note.

3.20.7 UDP TX Performance (ColdFire sending to UDP packets to host)

1. Start the PC UDP server on port 5678 (emg_udp_server.exe 5678)
2. Build and load the freescale_UDP_client target image.
3. Reset the board



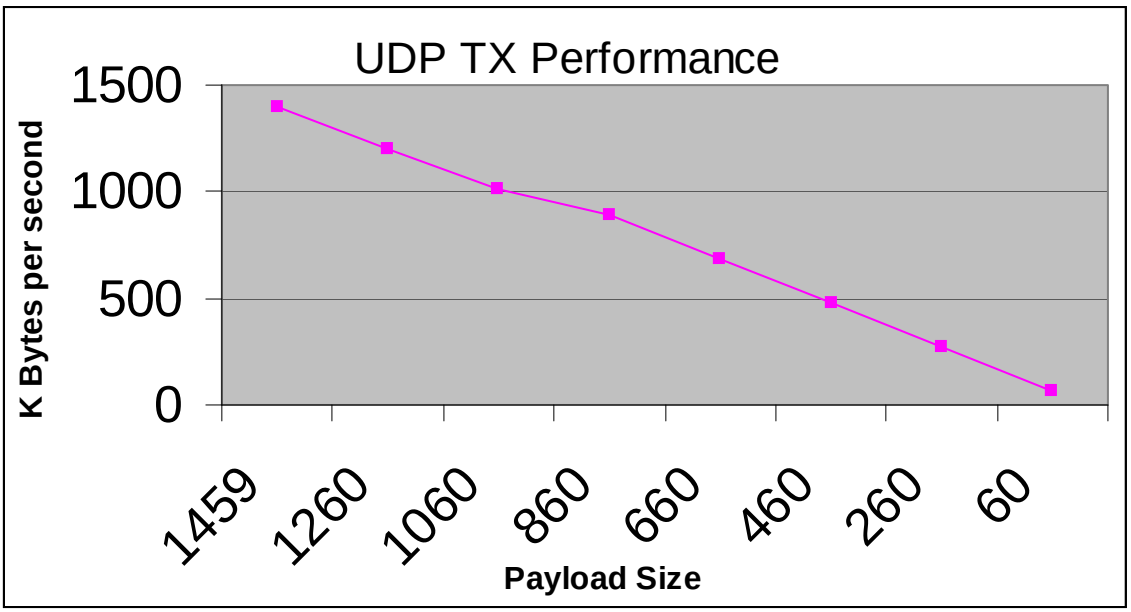


Figure 13. UDP TX Performance

3.20.8 UDP Client Firmware Used for Performance Testing

```
//*********************************************************************
//
// emg_udpsend based on tftp_udpsend
// Written by Eric Gregori
//
// Send outbuf to dest_port at ip address dest_ip
//
//*********************************************************************
int emg_udpsend (ip_addr dest_ip,
                                          unsigned short dest_port,
                                          char *outbuf,
                                          int outlen
                                    )
{
        PACKET           pkt2;   // packet to send & free
        int                       e;

        pkt2 = udp_alloc(outlen, 0);
        if(!pkt2)
                return -1;

        for( e=0; e<outlen; e++ )
                pkt2->nb_prot[e] = outbuf[e];

        pkt2->nb_plen = outlen;
        pkt2->fhost = dest_ip;
        pkt2->net         = NULL;
                                                        // local port
        e = udp_send( dest_port, 0x1234,     pkt2 );

   if(e < 0)
   {
#ifdef NPDEBUG
      dprintf("tftp_udpsend(): udp_send() error %d\n", e);
#endif
      return e;
   }
   else
      return 0;
}

//*********************************************************************
//
// Sample function for a UDP client
//
//*********************************************************************
void emg_send_data_via_udp( void )
{
        ip_addr          dest_ip = SERVER_IP;
        int              len;

        // THIS VALUE SETS THE PAYLOAD SIZE
        len = 1460;

        data_to_send[0] = data_to_send[0] + 1;
```

```
        // Send data_to_send to ip address det_ip, port PORT_NUMBER
        if( emg_udpsend( dest_ip, PORT_NUMBER, (void *)data_to_send, len ) )
                printf( "\nError sending via UDP " );
        printf( "\nsent %d", len );
}
```

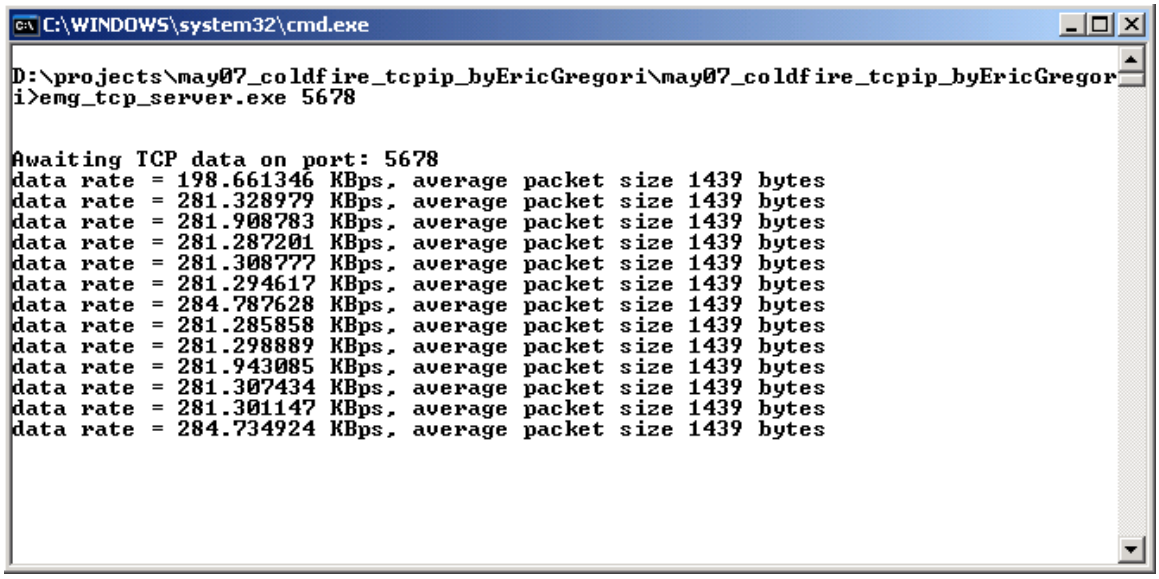
3.20.9 UDP Server PC Application Used for Performance Testing

```
//*********************************************************************
// UDP Server
//
// Written by: Eric Gregori
//*********************************************************************
void emg_udp_server( SOCKET s, struct sockaddr_in *peerp )
{
        int                                 rc, i, average;
        int                                 peerlen;
        double                      bytes_received, seconds;
        float                               rate;

        bytes_received = (double)0;
        reset_basetime();
        average = 0;
        while(1)
        {
                peerlen = sizeof( *peerp );
                rc = recvfrom( s, (char *)data, sizeof( data ), 0,
(struct sockaddr *)peerp, &peerlen );
                if( rc >= 0 )
                {
                        average += rc;
                        average /= 2;
                        bytes_received += (double)rc;
                        if( bytes_received >= (double)1000000 )
                        {
                                seconds = get_seconds();
                                rate = bytes_received/seconds;
                                printf( "\ndata rate = %f KBps, average packet
size %d", (rate/1024), average  );
                                bytes_received = (double)0;
                                reset_basetime();
                        }
                }
                else
                        error( 1, errno, "recvfrom failed" );
        }
}

//*********************************************************************
// UDP Server
//
// Written by: Eric Gregori
//*********************************************************************
int main ( int argc, char **argv )
{
    unsigned     __int64 pf;
        struct           sockaddr_inlocal;
```

```
        struct          sockaddr_inpeer;
        char                    *hname;
        int                     peerlen;
        int                     mode;
        SOCKET          s1;
        SOCKET          s;
        const                   char    on = 1;
        char                    port[256];

        INIT();

        hname = NULL;

        if( argc != 2 )
        {
                exit(0);
        }
        else
        {
                        strcpy( port, argv[1] );
                        mode = 2;
        }

        set_address( hname, port, &local, "udp" );
        s = socket( AF_INET, SOCK_DGRAM, 0 );
        if( !isvalidsock( s ) )
                error( 1, errno, "socket call failed" );

        if( bind( s, ( struct sockaddr * ) &local, sizeof( local ) ) )
                error( 1, errno, "bind failed" );

        reset_basetime();

        printf( "\n\nAwaiting UDP data on port: %s", port );

        emg_udp_server( s, &local );

        CLOSE( s );

        EXIT(0);
}
```

3.20.9.1 TCP TX Performance (ColdFire sending to TCP packets to host)

- Start the PC TCP server on port 5678 (emg_tcp_server.exe 5678)
- Build and load the freescale_TCP_client target image.
- Reset the board



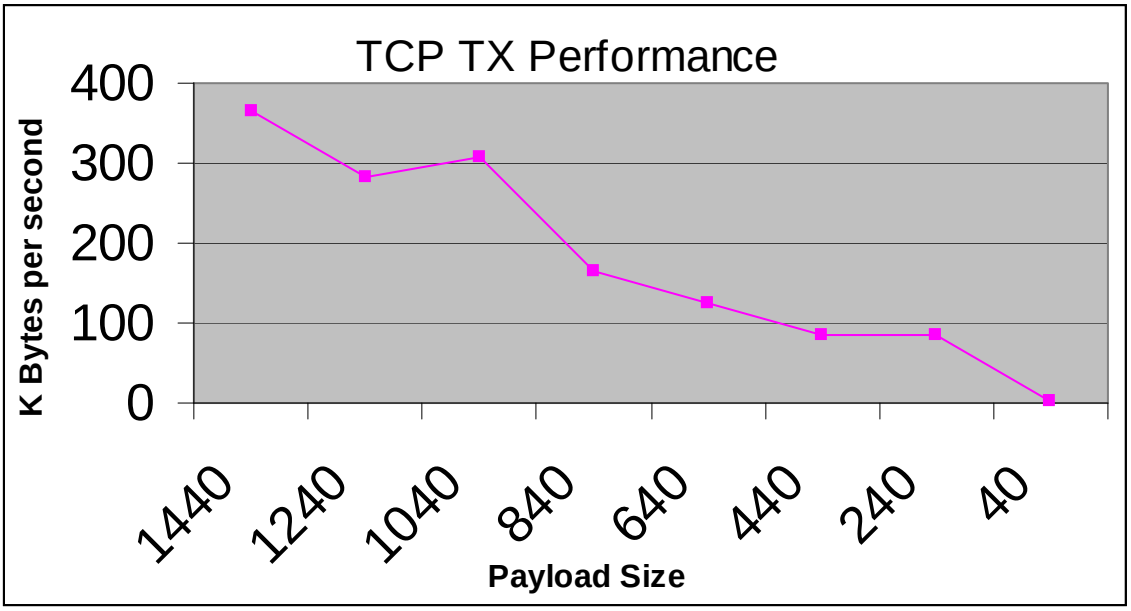


Figure 14. TCP TX Performance

3.20.10 TCP Client Firmware Used for Performance Testing

```
//*********************************************************************
// emg_tcp_tx()
//
// Take data from the UART RX buffer and send it out over
// ethernet using the m_send() function.
//*********************************************************************
void emg_tcp_tx( void )
{
        int             len;

        len = 1440;   // ADJUST THIS FOR PAYLOAD LENGTH
```

```
        (void)m_send( emg_tcp_communication_socket, (char *)data_to_send, len );
        printf( "\nsent %d", len );
}


//*********************************************************************
// emg_tcp_loop() -  Written By Eric Gregori
//                                          eric.gregori@freescale.com
//
// Run application
//
//*********************************************************************
int freescale_tcp_loop()
{
        int                     i;


        emg_tcp_tx();
        tk_sleep( INTER_PACKET_DELAY );
        return SUCCESS;
}


//*********************************************************************
// emg_http_cmdcb() - Written by Eric Gregori
//                                           eric.gregori@freescale.com
//
// This is the mini-sockets callback function.
// We only uses to detect a connection to the socket.
// When a connection is made, this function is called by the stack.
// The connection message is sent to the application through a
// msring queue.
//*********************************************************************
int freescale_tcp_cmdcb(int code, M_SOCK so, void * data)
{
        int e = 0;

        switch(code)
        {
                // socket open complete
                case M_OPENOK:
                        msring_add(&emg_tcp_msring, so);
                        break;

                // socket has closed
                case M_CLOSED:
                        while( semaphore ){};
                        semaphore = 1;
                        emg_tcp_communication_socket = INVALID_SOCKET;
                        semaphore = 0;
                break;

                // passing received data
                // blocked transmit now ready
                case M_RXDATA:      // received data packet, let recv() handle it
                case M_TXDATA:      // ready to send more, loop will do it
```

```
                        e = -1;        // return nonzero code to indicate we don't want it
                        break;

                default:
                dtrap();                          // not a legal case
                return 0;
   }

   TK_WAKE(&to_emgtcpsrv);         // wake server task
   USE_VOID(data);
   return e;
}


//*****************************************************************
// emg_tcp_init() - written by Eric Gregori
//                                        eric.gregori@freescale.com
//
// Create and bind a socket to our listening port ( PORT_NUMBER ).
// Set the socket to listen and non-blocking.
//*****************************************************************
int freescale_tcp_init()
{
        int e;


        semaphore                                          = 0;
        flash_ffs_lockout                  = 0;
        emg_tcp_communication_socket = INVALID_SOCKET;

        // Init message queue for MINI_TCP socket interface
    msring_init(&emg_tcp_msring, emg_tcp_msring_buf, sizeof(emg_tcp_msring_buf) /
sizeof(emg_tcp_msring_buf[0]));

        // Init a socket structure with our Port Number
        emg_tcp_sin.sin_addr.s_addr = (SERVER_IP);
        emg_tcp_sin.sin_port        = (PORT_NUMBER);

        emg_tcp_communication_socket= m_socket();

        printf( "\nConnecting to target" );

        // Socket is blocking.  The m_connect call will block
        // until it connects.
    e = m_connect(emg_tcp_communication_socket, &emg_tcp_sin, freescale_tcp_cmdcb );
    if( e > 0 )
    {
        if( e == ECONNREFUSED )
                printf( " - Cold Not Find Target, reset" );
        else
                printf(" - error %d starting listen on emg TCP server\n", e);

        while(1)
                tk_sleep(100);
    }

        emg_tcp_server_socket = emg_tcp_communication_socket;
```

```
        printf( " - Connected" );

        for( e=0; e<TEST_BUFFER/4; e++ )
                data_to_send[e] = e;

        return SUCCESS ;
}


//*********************************************************************
// emg_tcp_check() -  Written by Eric Gregori
//                                          eric.gregori@freescale.com
//
// Check msring for message from socket callback function.
// If we received a connect request, call the connection function.
// While we are waiting for a connection to complete, we need to
// continue running our loop.
// Make the socket non-blocking.
//
// Call the loop function to execute any pending sessions.
//*********************************************************************
void freescale_tcp_check(void)
{
        M_SOCK          so;


        if ( emg_tcp_server_socket == INVALID_SOCKET )
        return ;

        if( emg_tcp_communication_socket != INVALID_SOCKET )
                freescale_tcp_loop();
}


//*********************************************************************
// The application thread works on a "controlled polling" basis:
// it wakes up periodically and polls for work.
//
// The task could alternatively be set up to use blocking sockets,
// in which case the loops below would only call the "xxx_check()"
// routines - suspending would be handled by the TCP code.
//
//
// FUNCTION: tk_emg_tcp_srv
//
// PARAM1: n/a
//
// RETURNS: n/a
//
//*********************************************************************
TK_ENTRY(tk_emgtcpsrv)
{
   int err;

   while (!iniche_net_ready)
      TK_SLEEP(1);
```

```
   err = freescale_tcp_init();
   if( err == SUCCESS )
   {
      exit_hook(freescale_tcp_cleanup);
   }
   else
   {
      dtrap();                                          // emghttp_init()  shouldn't ever fail
   }

   for (;;)
   {
      freescale_tcp_check();         // will block on select
      tk_yield();                   // give up CPU in case it didn't block

      if (net_system_exit)
         break;
   }
   TK_RETURN_OK();
}


//*********************************************************************
// create_freescale_task() - Written by Eric Gregori
//                                                          eric.gregori@freescale.com
//
// Insert the FreeScale task into the RTOS.
//*********************************************************************
void create_freescale_task( void )
{
        int e = 0;

        e = TK_NEWTASK(&emg_tcp_task);
        if (e != 0)
        {
      dprintf("freescale task create error\n");
      panic("create_apptasks");
        }
}
```

3.20.11 TCP Server PC Application Used for Performance Testing

```
//*********************************************************************
// init
//
// Initialize the Winsock DLL ( Windows specific )
//
// Written by: Eric Gregori
//*********************************************************************
void init( char **argv )
{
        WSADATA           wsadata;

        ( program_name = strrchr( argv[0], '\\' )) ? program_name++ :
        (program_name = argv[0] );
        WSAStartup( MAKEWORD( 2,2 ), &wsadata );
```

```
}


void emg_tcp_server( SOCKET s, struct sockaddr_in *peerp )
{
        int                                 rc, i, average;
        int                                 peerlen;
        double                      bytes_received, seconds;
        float                       rate;

        bytes_received = (double)0;
        reset_basetime();
        average = 0;
        while(1)
        {
                peerlen = sizeof( *peerp );
                rc = recv( s, (char *)data, sizeof( data ), 0 );
                if( rc > 0 )
                {
                        average += rc;
                        average /= 2;
                        bytes_received += (double)rc;
                        if( bytes_received >= (double)100000 )
                        {
                                seconds = get_seconds();
                                rate = bytes_received/seconds;
                                printf( "\ndata rate = %f KBps, average packet size %d
bytes", (rate/1024), average  );
                                bytes_received = (double)0;
                                reset_basetime();
                                average = 0;
                        }
                }
                else
                        error( 1, errno, "recv failed" );
        }
}

//*****************************************************************
// TCP Server
//
// Written by: Eric Gregori
//*****************************************************************
int main ( int argc, char **argv )
{
    unsigned__int64 pf;
        struct          sockaddr_inlocal;
        struct          sockaddr_inpeer;
        char            *hname;
        int                     peerlen;
        int                     mode;
        SOCKET          s1;
        SOCKET          s;
        const           char    on = 1;
        char            port[256];

        INIT();
```

```
        hname = NULL;

        if( argc != 2 )
        {
                exit(0);
        }
        else
        {
                if( (strcmp( argv[1], "HTTP" ) == 0) ||
(strcmp( argv[1], "http" ) == 0) )
                {
                        strcpy( port, "80" );
                        mode = 1;
                }
                else
                {
                        strcpy( port, argv[1] );
                        mode = 2;
                }
        }

        set_address( hname, port, &local, "tcp" );
        s = socket( AF_INET, SOCK_STREAM, 0 );
        if( !isvalidsock( s ) )
                error( 1, errno, "socket call failed" );

        if( setsockopt( s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof( on )))
                error( 1, errno, "setsockopt failed" );

        if( bind( s, ( struct sockaddr * ) &local, sizeof( local ) ) )
                error( 1, errno, "bind failed" );

        if( listen( s, 5 ) )
                error( 1, errno, "listen failed" );

        printf( "\n\nAwaiting TCP data on port: %s", port );

        do
        {
                peerlen = sizeof( peer );
                s1 = accept( s, (struct sockaddr *)&peer, &peerlen );
                if( !isvalidsock( s1 ) )
                        error( 1, errno, "accept failed" );

                reset_basetime();
                emg_tcp_server( s1, &local );

                CLOSE( s1 );
        } while( 1 );


        CLOSE( s );

        EXIT(0);
}
```

3.20.12 Porting the Stack and RTOS to Other ColdFire Processors

The stack is a black box that requires inputs and support for outputs. When porting the stack to other ColdFire processors, the goal is to provide the inputs and outputs. The stack itself is a group of files that need to be compiled and linked.

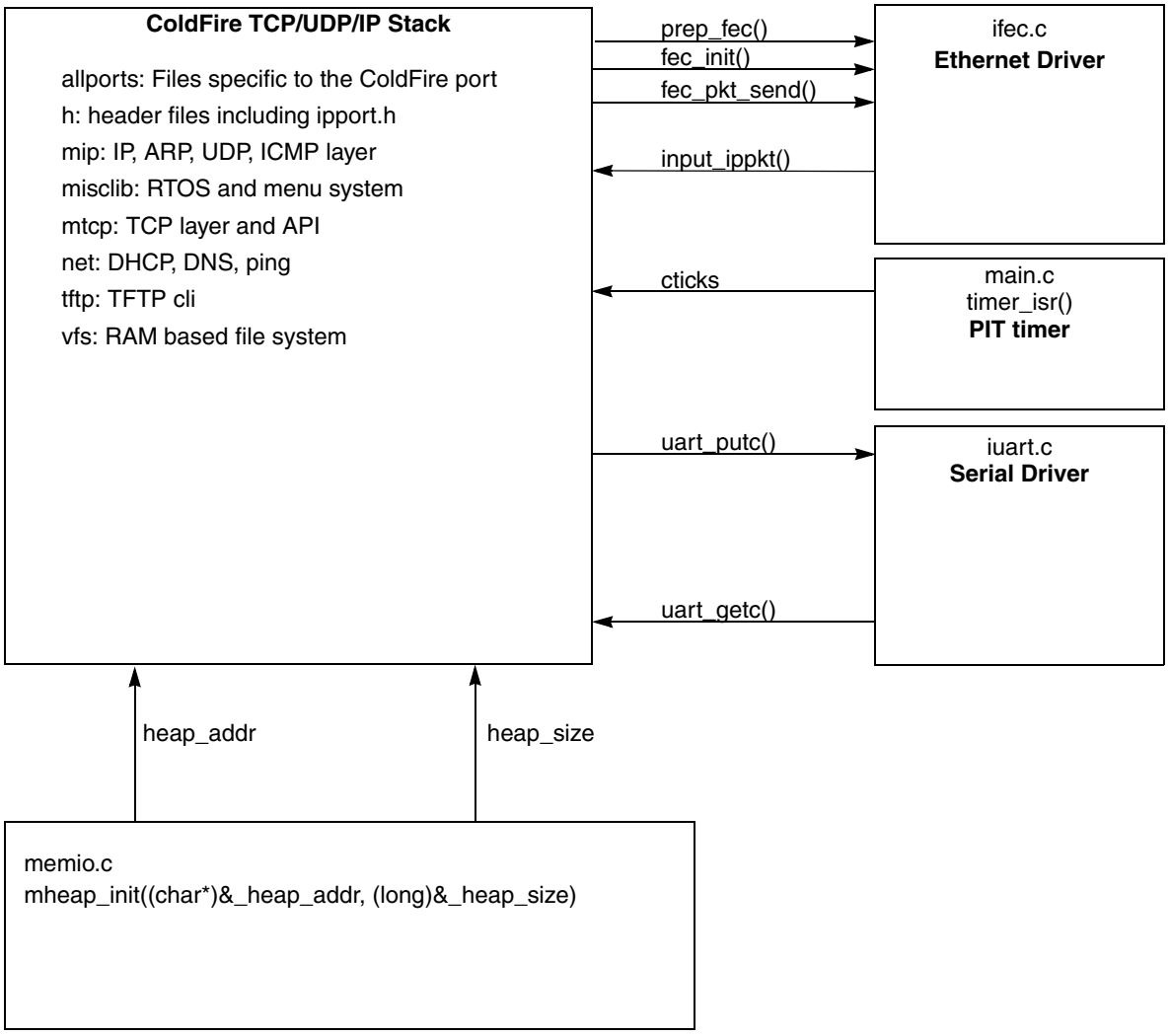


Figure 15. ColdFire TCP/UDP/IP Stack

3.20.13 Periodic Timer

The stack requires a periodic timer to keep track of events and timeouts. Since the RTOS is not a pre-emptive RTOS, there is no system tick. Task switching occurs asynchronously. The periodic timer must increment a variable (cticks) defined in main.c. The stack references all timing of events off of cticks. Cticks is an unsigned long that is not reset, only a rollover. If the RTOS is used, sleep() is in cticks. This version of the port uses a ColdFire PIT as the interval timer. The PIT creates a interrupt every 1 ms, calling the timer_isr() in main.c to increment cticks every 5 ms.

3.20.14 MCF5223X Clock Configuration

The MCF5223X has an internal PHY, so it requires that the external clock be 25 Mhz. Since 25 Mhz is too high of a frequency to input directly into the PLL, it must first be divided by a prescaler. With the MCF5223X, this is accomplished using the CCHR register. Older revisions of the MCF5223X reference manual incorrectly state that the CCHR resets to 0. This would result in a divide by 1, and would be outside the spec for the MCF5223X PLL. In fact, the CCHR resets to a value of 4, resulting in a divide by 5. This results in the external 25 Mhz clock being divided by 5 before entering the PLL.

The stack is configured to run at 60 Mhz on the MCF5223X. This is done in the function mcf5223_pll_init(void) in mcf5223_sysinit.c. The PLL is configured to multiply the 5 Mhz by 12 via the SYNCR register, resulting in the PLL outputting 60 MHz.

```
// EMG - The CCHR resets to a divide by 5, resulting in PLL input = 5 MHz
MCF_CLOCK_SYNCR = MCF_CLOCK_SYNCR_MFD(4)  |
                                  MCF_CLOCK_SYNCR_CLKSRC |
                                  MCF_CLOCK_SYNCR_PLLMODE |
                                  MCF_CLOCK_SYNCR_PLLEN ;

// MCF_CLOCK_SYNCR_MFD(4) = PLL in ( 5 MHz ) * 12
// MCF_CLOCK_SYNCR_CLKSRC = PLL output drives the system clock
// MCF_CLOCK_SYNCR_PLLMODE = PLL is in Normal mode
// MCF_CLOCK_SYNCR_PLLEN = PLL enabled
```

3.20.15 MCF5223X PIT Configuration

The stack requires that the timer_isr() in main.c be called every PIT1_INTS_PER_SEC (defined in main.c). With the MCF5223X, we chose a PIT to tickle the timer isr. The timer isr then increments cticks every INTS_PER_TICK (defined in main.c).

```
PIT1_INTS_PER_SEC = 1000  ( 1ms )
CTICKS_PER_SEC = 200 ( 5ms )

INTS_PER_CTICK = PIT1_INTS_PER_SEC / CTICKS_PER_SEC = 5
```

The PIT timer is configured in the function PIT_Timer_Init (uint8 PCSR, uint16 PMR) in mcf5223.c. This function is called from pre_task_setup() in iutil.c, and pre_task_setup() is called from netmain_init() in allports.c.

```
char *pre_task_setup()
{
   PIT_Timer_Init(0,30000);
   return NULL;
}
```

- Where 0 is the PCSR and 30000 is the PMR.
- For a PIT, the interrupt interval formula is PMR / ((Fsys/2)/(2^PCSR))
- The peripherals in the MCF5223 are driven by Fsys/2. This may not be clear in the reference manual.
- Fsys = 60 Mhz -> interval = 30000/((60000000/2)/(2^0)) = .001 = 1ms

3.20.16 Ethernet Controller

If the stack is used for ethernet communications, an ethernet controller and driver must be supplied. A SLIP driver is available for non-ethernet applications. Contact the field applications engineer. The port of the stack that accompanies this application note includes a driver for the ColdFire Fast Ethernet Controller (FEC)in ifec.c. The FEC is a DMA-based ethernet controller that supports 10 Mbps and 100 Mbps. Data is passed back and forth between RAM and the FEC via DMA transactions controlled by buffer descriptors. Each buffer descriptor consists of four 16 bit words. The first word is a configuration/status word, the second word is the buffer length, and the third and fourth words combine to form a pointer to a buffer. Buffer descriptors are configured in a ring, with the wrap bit set in the configuration word to indicate the last descriptor.

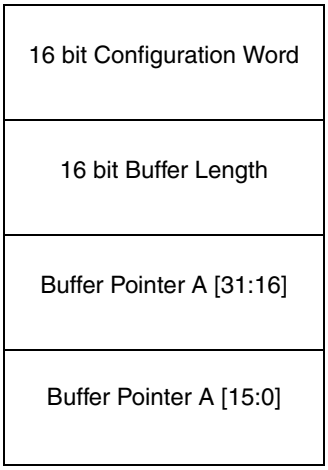


Figure 16. Buffer Descriptor

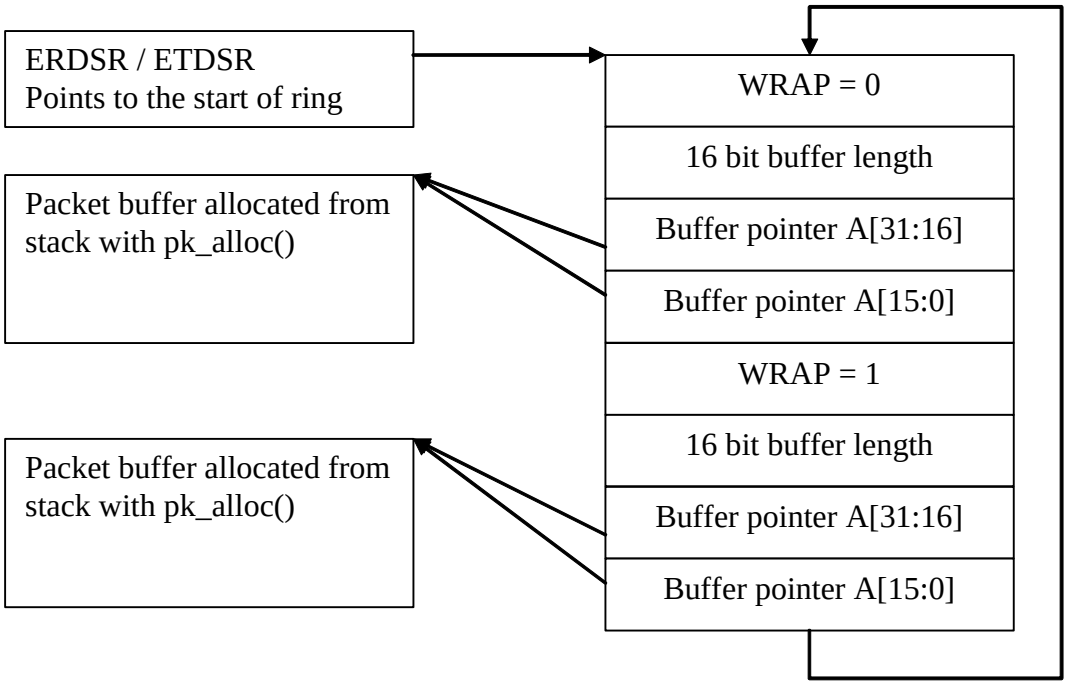


Figure 17. Buffer Descriptor Ring

4 The Ethernet PHY

An ethernet (IEEE802.3) system has two parts, the Media Access Layer (MAC) and the Physical Layer (PHY). The MAC performs the protocol layer while the PHY performs the electrical layer. There are two types of PHY (there are actually many more but the ColdFire currently supports only two) 100 Mbps (IEEE802.3u) and 10 Mbps (IEEE802.3a-t). Each PHY is essentially an electrical modulator. Although the PHYs are different, in most cases they are implemented in a single part.

The PHY communicates with the MAC via a Media Independent Interface (MII) (IEEE802.3u). Part of the MII is a serial Management Interface (in FEC documentation this is referred to as the Media Management Interface). This management interface grants access to configuration registers in the PHY. The MSCR and MMFR registers in the FEC are used to init and write to/read from the Media Management Interface. The MSCR register is used to configure the clock rate of the serial data transfer. The 802.3u specification limits this rate to 2.5 MHz.

4.1 Initializing MII Interface in MCF5223X (in mii.c)

```
/*********************************************************************/
 * Initialize the MII interface controller
 * Parameters:
 *  sys_clk System Clock Frequency (in MHz)
/*********************************************************************/
void fec_mii_init(int sys_clk)
```

```
{
    /*
     * Initialize the MII clock (EMDC) frequency
     *
     * Desired MII clock is 2.5 MHz
     * MII Speed Setting = System_Clock / (2.5 MHz * 2)
     * (plus 1 to make sure we round up)
     */
    MCF_FEC_MSCR = MCF_FEC_MSCR_MII_SPEED((uint32)((sys_clk/3)+1));
    /*
     * Make sure the external MII interface signals are enabled
     */
// not needed on 5223
//       MCF_GPIO_PASPAR = (MCF_GPIO_PASPAR & 0xF0FF) | 0x0F00;
}
```

4.2 MII Management Frame Write Function (in mii.c)

After the speed is initialized, data is read and written to the PHY registers using the MMFR register. The MII specification actually supports multiple PHYs on an MII, so each PHY is given an address. The FEC does all the work. Specify a PHY address, register address, and data, and the FEC handles everything else.

```
/********************************************************************/
* Write a value to a PHY's MII register.
 *
 * Parameters:
 *  phy_addr    Address of the PHY.
 *  reg_addr    Address of the register in the PHY.
 *  data        Data to be written to the PHY register.
 *
 * Return Values:
 *  0 on failure
 *  1 on success.
 *
/********************************************************************/
int fec_mii_write(int phy_addr, int reg_addr, int data)
{
    int timeout;
    uint32 eimr;

    /* Write to the MII Management Frame Register to kick-off the MII write */
    MCF_FEC_MMFR = (vuint32)(0
        | MCF_FEC_MMFR_ST_01
        | MCF_FEC_MMFR_OP_WRITE
        | MCF_FEC_MMFR_PA(phy_addr)
        | MCF_FEC_MMFR_RA(reg_addr)
        | MCF_FEC_MMFR_TA_10
        | MCF_FEC_MMFR_DATA(data));

    /* Poll for the MII interrupt (interrupt should be masked) */
    for (timeout = 0; timeout < FEC_MII_TIMEOUT; timeout++)
    {
        if (MCF_FEC_EIR & MCF_FEC_EIR_MII)
            break;
    }
    if(timeout == FEC_MII_TIMEOUT)
        return 0;
```

```
    return 1;
}
```

4.3 MII Management Frame Read Function (in mii.c)

```
/********************************************************************/
* Read a value from a PHY's MII register.
 *
 * Parameters:
 *  phy_addr    Address of the PHY.
 *  reg_addr    Address of the register in the PHY.
 *  data        Pointer to storage for the Data to be read
 *              from the PHY register (passed by reference)
 *
 * Return Values:
 *  0 on failure
 *  1 on success.
 *
/********************************************************************/
int fec_mii_read(int phy_addr, int reg_addr, uint16* data)
{
    int timeout;
    uint32 eimr;

    /* Write to the MII Management Frame Register to kick-off the MII read */
    MCF_FEC_MMFR = (vuint32)(0
        | MCF_FEC_MMFR_ST_01
        | MCF_FEC_MMFR_OP_READ
        | MCF_FEC_MMFR_PA(phy_addr)
        | MCF_FEC_MMFR_RA(reg_addr)
        | MCF_FEC_MMFR_TA_10);

    /* Poll for the MII interrupt (interrupt should be masked) */
    for (timeout = 0; timeout < FEC_MII_TIMEOUT; timeout++)
    {
        if (MCF_FEC_EIR & MCF_FEC_EIR_MII)
            break;
    }

    if(timeout == FEC_MII_TIMEOUT)
        return 0;

    *data = (uint16)(MCF_FEC_MMFR & 0x0000FFFF);

    return 1;
}
```

The PHY registers themselves depend on the PHY part being used.

Table 1. MCF5223x PHY MII Registers

Address		Use	Access
0	$0X0000	Control Register	Read/Write
1	$0X0001	Status Register	Read/Write
2	$0X0002	PHY identification Register 1	Read/Write
3	$0X0003	PHY Identification Register 2	Read/Write
4	$0X0004	Auto-Negotiation Advertisement Register	Read/Write
5	$0X0005	Auto-Negotiation Link Parter Ability Register	Read/Write
6	$0X0006	Auto-Negotiation Expansion Register	Read/Write
7	$0X0007	Auto-Negotiation Next Page Register	Read/Write
8	$0X0008	RESERVED	Read/Write
9	$0X0009	RESERVED	Read/Write
10	$0X000A	RESERVED	Read/Write
11	$0X000B	RESERVED	Read/Write
12	$0X000C	RESERVED	Read/Write
13	$0X000D	RESERVED	Read/Write
14	$0X000E	RESERVED	Read/Write
15	$0X000F	RESERVED	Read/Write
16	$0X0010	Interrupt Control Register	Read/Write
17	$0X0011	Proprietary Status Register	Read/Write
18	$0X0012	Proprietary Control Register	Read/Write

4.4 Media Management Interface (in menulib.c)

```
//*********************************************************************
// int SoftEthernetNegotiation( int seconds )   Written By Eric Gregori
//
// Attempt to connect at 100 Mbps - Half Duplexe
// Wait for seconds
// Attempt to connect at 10 Mbps - Half Duplexe
// Returns 10, or 100 on success, 0 on failure
//*********************************************************************
int SoftEthernetNegotiation( int seconds )
{
        uint16          reg0, reg1, tick;

        // Force ePHY to manual, 100mbps, Half Duplexe
        (void)fec_mii_read(0, 0, &reg0);
        reg0 |= 0x2000;                     // 100Mbps
        reg0 &= ~0x0100;                    // Half Duplexe
        reg0 &= ~0x1000;                    // Manual Mode
```

```
        (void)fec_mii_write( 0, 0, reg0 );
        (void)fec_mii_write( 0, 0, (reg0|0x0200) );   // Force re-negotiate
        for( tick=0, ephy_isr=0; tick<100; tick++ )
        {
                tk_sleep( seconds*10 );
                (void)fec_mii_read(0, 1, &reg1);
                if( reg1 & 0x0004 )
                        return(100);
        }
        // Force ePHY to manual, 10mbps, Half Duplexe
        (void)fec_mii_read(0, 0, &reg0);
        reg0 &= ~0x2000;                    // 10Mbps
        reg0 &= ~0x0100;                    // Half Duplexe
        reg0 &= ~0x1000;                    // Manual Mode
        (void)fec_mii_write( 0, 0, reg0 );
        (void)fec_mii_write( 0, 0, (reg0|0x0200) );   // Force re-negotiate
        for( tick=0, ephy_isr=0; tick<100; tick++ )
        {
                tk_sleep( seconds*10 );
                (void)fec_mii_read(0, 1, &reg1);
                if( reg1 & 0x0004 )
                        return(10);
        }
        return(0);
}
```

In most cases the PHY does not need to be configured. The PHY on the M5329EVB is bootstrapped to advertise support for 10/100 Half and Full duplex with autonegotiations enabled. Advertising refers to the capabilities a PHY announces it has during autonegotiations with another PHY. Autonegotiations is a PHY-to-PHY protocol defined in 802.3u. In the protocol, each PHY advertises it abilities, and the two PHYs negotiate a set of parameters they can both support. Since the PHY on the M5329EVB defaults to normal operating parameters, we do not need to configure it. The M5208EVB uses the same PHY and bootstrap configurations, so the same is true with a port to the M5208EVB. If the PHY registers do need to be modified, include the mii.c and mii.h files in the port.

See the the ifstats() function for further information.

5 Porting the ColdFire TCP/UDP/IP Stack Project Using CodeWarrior™

The stack has been written to be easily ported between ColdFire platforms. The source code for the stack can be easily incorporated into any stationary created by CodeWarrior for a ColdFire project. The stack was originally written for the MCF5208, then ported to the MCF5223X, then ported to the MCF5282, and with this application note ported to the MCF523X.

1. Using CodeWarrior stationary create a C project
2. Add the files from the following stack directories to the new project:
   Ethernet – The mii driver may not be needed if using default PHY.
   Common
   Projects
3. Modify common.h per instructions below.
4. Remove stack version of startup.c (it's for a MCF5223X)
5. If the ColdFire does not have internal flash (CFM). Remove freescale_flash_loader.c and replace with stubs.

6. If the ColdFire does not have an SPI. Remove Freescale_serial_flash.c
7. Add entries in the vector table for:
    a. ethernet_handler
    b. timer_isr
    c. uart0_isr
    d. fec_isr
8. Modify interrupt init functions for above handlers via macro
9. Enable port pin functionality for FEC.
10. Enable/Disable any stack features via ipport.h

5.1 Modifying common.h from New Project For Stack Usage

Common.h must be modified to disable standard library usage. Also the common.h included with the stack should not be included with your new project (use the common.h that was automatically created when you started a new project). The CodeWarrior stationary will put the common.h file in the project include directory.

Bold lines have been modified from the original stationary generated common.h.

```
#ifndef _COMMON_H_
#define _COMMON_H_

/*********************************************************************/
//* Debug prints ON (#define) or OFF (#undef)
//#define DEBUG_PRINT

 //* Include the generic CPU header file
#include "mcf5xxx.h"

 //* Include the specific CPU header file
#include "mcf532x.h"

 //* Include the board specific header file
#include "m5329evb.h"

 //* Include any toolchain specific header files
//#include "mwerks.h"

#if (defined(__MWERKS__))
#include "mwerks.h"
#define __CFM68K__ 0
#define __MC68K__ 0
#elif (defined(__DCC__))
#include "build/wrs/diab.h"
#elif (defined(__ghs__))
#include "build/ghs/ghs.h"
#endif

/*
 * Include common utilities
 */
// EMG #include <assert.h>
// EMG #include <stdlib.h>

/*********************************************************************/
#endif /* _COMMON_H_ */
```

THIS PAGE IS INTENTIONALLY BLANK

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3470
Rev. 0
06/2007

freescale™
semiconductor