Freescale Semiconductor
Application Note

AN3518
Rev. 0, 09/2007

Advanced ColdFire TCP/IP Clients

HTTP, DNS, XML, and RSS R0.9

by: Eric Gregori

Contents

1 Introduction

This application note discusses advanced topics in ColdFire client-side TCP/IP firmware design. Client designs minimize RAM usage, while optimizing performance. The HTTP client firmware described in this document runs on top of the ColdFire TCP/IP stack. The DNS client included in the stack is also covered. It is used by the HTTP server for domain name transalation.

The second portion of this document covers an interesting use of the HTTP client: a RSS feed reader. Using the RSS feed reader on top of the HTTP client, you can read and decode RSS feeds and other XML documents. The RSS feed reader section also includes a section on interfacing a parallel LCD to the ColdFire processor. The RSS feed data can be output to the serial port or scrolled on the LCD display.

2 HTTP Protocol

Hyper-Text Transport Protocol (HTTP) is the communication protocol of the world wide web. HTTP is used to transfer web pages (Hyper-Text documents) across the internet. An HTTP connection consists of two parts: the HTTP client (web browser) and the HTTP server. You can use the HTTP client to receive and view the web page and the HTTP server to store, organize, and transfer the web pages.

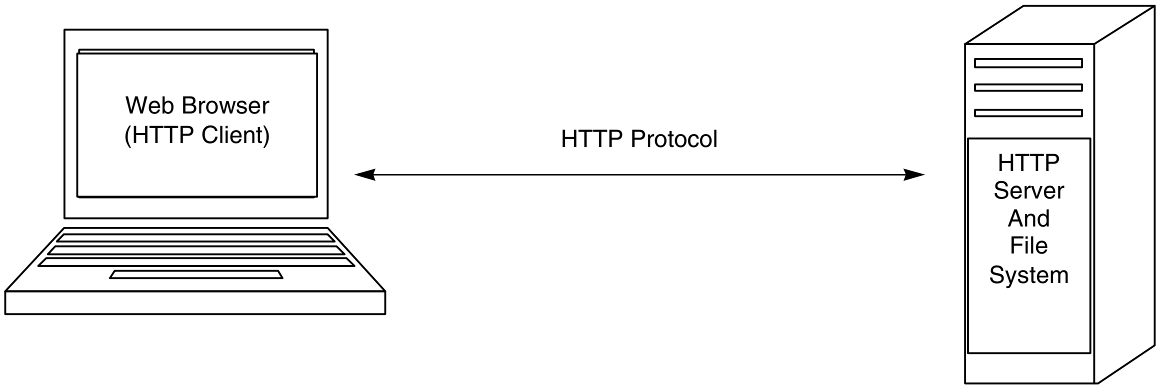


Figure 1. Basic HTTP Block Diagram

HTTP is defined by technical specifications RFC2616 (HTTP version 1.1) and RFC1945 (HTTP version 1.0). RFCs are published by the Internet Engineering Task Force (IETF). See http://www.rfc-editor.org for more information.

HTTP is a request response protocol. The client requests a web page from the server and the server responds with the web page contents (HTML — Hyper-Text Markup Language). HTTP can be used to send any type of data, including binary data. The client requests a file using the GET method (HTTP is an ASCII protocol). The server responds with an HTTP header followed by the file contents. The client can also send a file using a POST method. Within the request, the HTTP version is embedded in ASCII. This notifies the server of the limitations of the client.

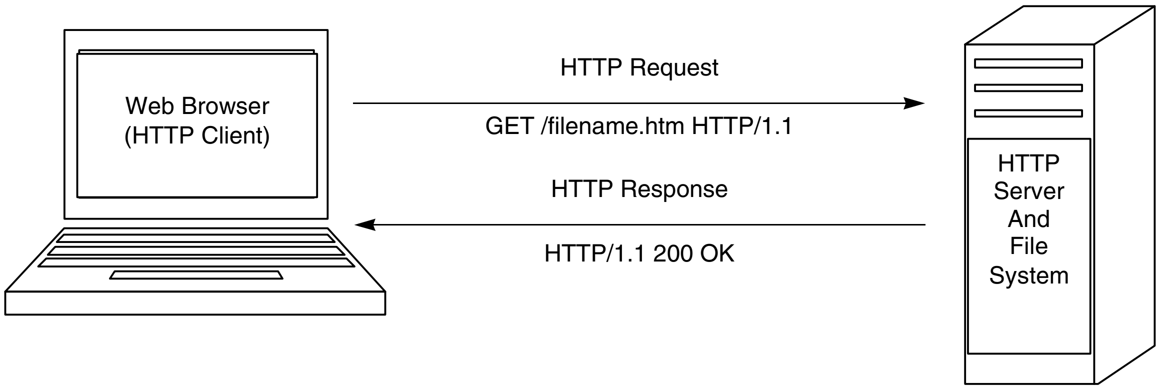

Web Browser (HTTP Client)
HTTP Request
GET /filename.htm HTTP/1.1
HTTP Response
HTTP/1.1 200 OK
HTTP Server And File System

Figure 2. Basic HTTP Request/Response Block Diagram

2.1 HTTP Request Example

The following is sent from the client (web browser) to the HTTP server:

```
GET /filename.htm HTTP/1.1
    # Asks the server to respond with the contents of filename.htm
    # Tells the server that it supports the HTTP1.1 standard
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword
    # Tells the server that it supports: gif, x-xbitmaps, jpeg, and pjpeg images,
    #   and msword documents
Accept-language: en-us
    # Tells the server that the language is English
Accept-Encoding: gzip, deflate
    # The gzip and deflate decompression algorithms are available
User-Agent: Mozzilla/4.0 (compatable; MSIE 6.0; Windows NT 5.1)
    # Tells the server that the browser is running IE6.0 on a Windows computer
Connection: Keep-Alive
    # Tells the server not to close the connection after the file is sent
```

2.2 HTTP Methods

The GET method is one method supported by RFC2616. Other methods are listed in Table 1.

Table 1. RFC2616 Methods

Method	RFC2616 Location	Description
Options	Section 9.2	Request for information
Get	Section 9.3	Request a file or data
Head	Section 9.4	Identical to GET without a message-body in the response
Post	Section 9.5	Send data
Put	Section 9.6	Send a file
Delete	Section 9.7	Delete a file
Trace	Section 9.8	Echo request
Connect	Section 9.9	Reserved for tunneling

2.3 HTTP Response Example

The server responds to the GET method with the following header:

```
HTTP/1.1 200 OK
    # Tells the client/browser that HTTP1.1 is supported, and the 200 status code tells
    #   the client that the file was found
Server: Microsoft-IIS/6.0
    # Informs the client of the web server type and version
Cache-Control: no-cache
    # Tells the client to disable cache
Content-Type: text/html
    # Tells the client the type of data that follows
Content-Encoding: gzip
    # Tells the client that the following data is encrypted using gzip
```

```
      Content-Length: 9062
          # Tells the client how many bytes are to follow
```

Followed by data from file, in this case encoded using gzip

2.4 Connection Persistance

Connection persistence, or KEEP ALIVE, is a protocol feature used to increase performance by decreasing TCP/IP overhead. A normal non-persistent HTTP transaction consists of:

- A TCP/IP connect
- GET method
- File transfer
- A TCP/IP close

This process is followed for every file the client needs (often multiple times with a single web page). The TCP/IP overhead takes a significant amount of time. With a persistent connection, the TCP/IP connect only occurs before the first file transfer and the TCP/IP connection is not closed after the file transfer. Instead the server goes into a status to wait for another method (GET).

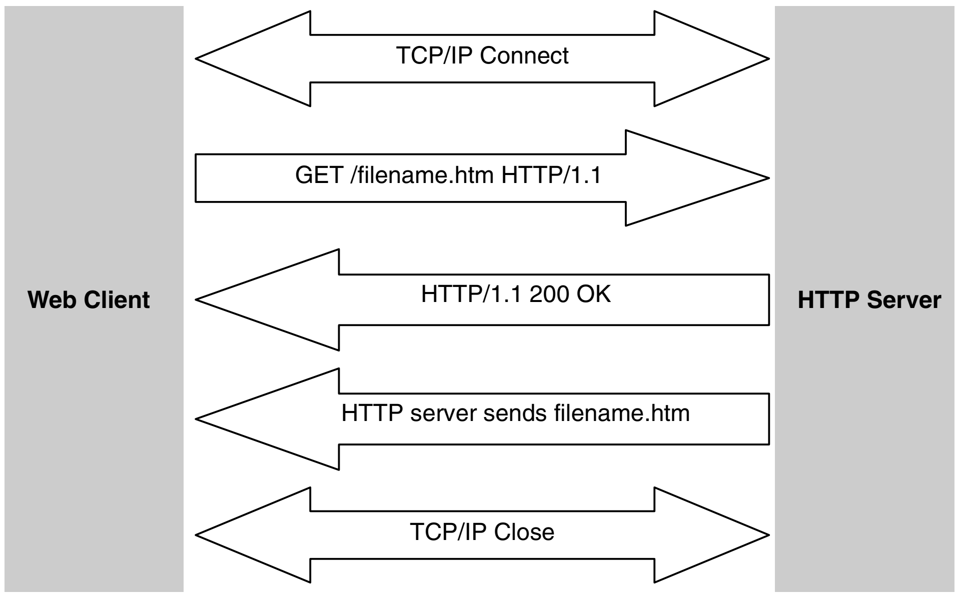


Figure 3. Non-Persistant HTTP Transaction

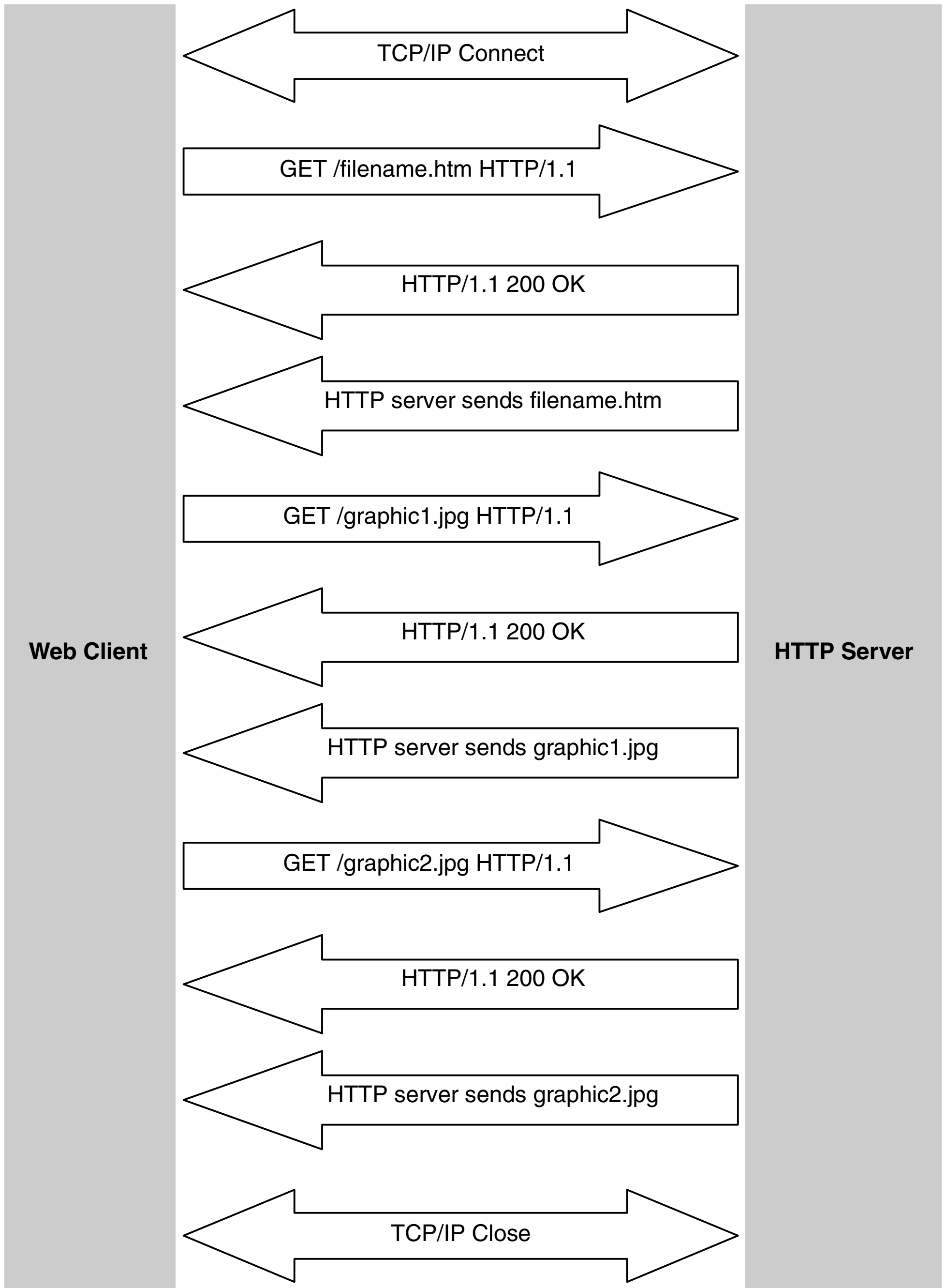

Web Client
TCP/IP Connect
GET /filename.htm HTTP/1.1
HTTP/1.1 200 OK
HTTP server sends filename.htm
GET /graphic1.jpg HTTP/1.1
HTTP/1.1 200 OK
HTTP server sends graphic1.jpg
GET /graphic2.jpg HTTP/1.1
HTTP/1.1 200 OK
HTTP server sends graphic2.jpg
TCP/IP Close
HTTP Server

Figure 4. Persistant HTTP Transaction

3 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.

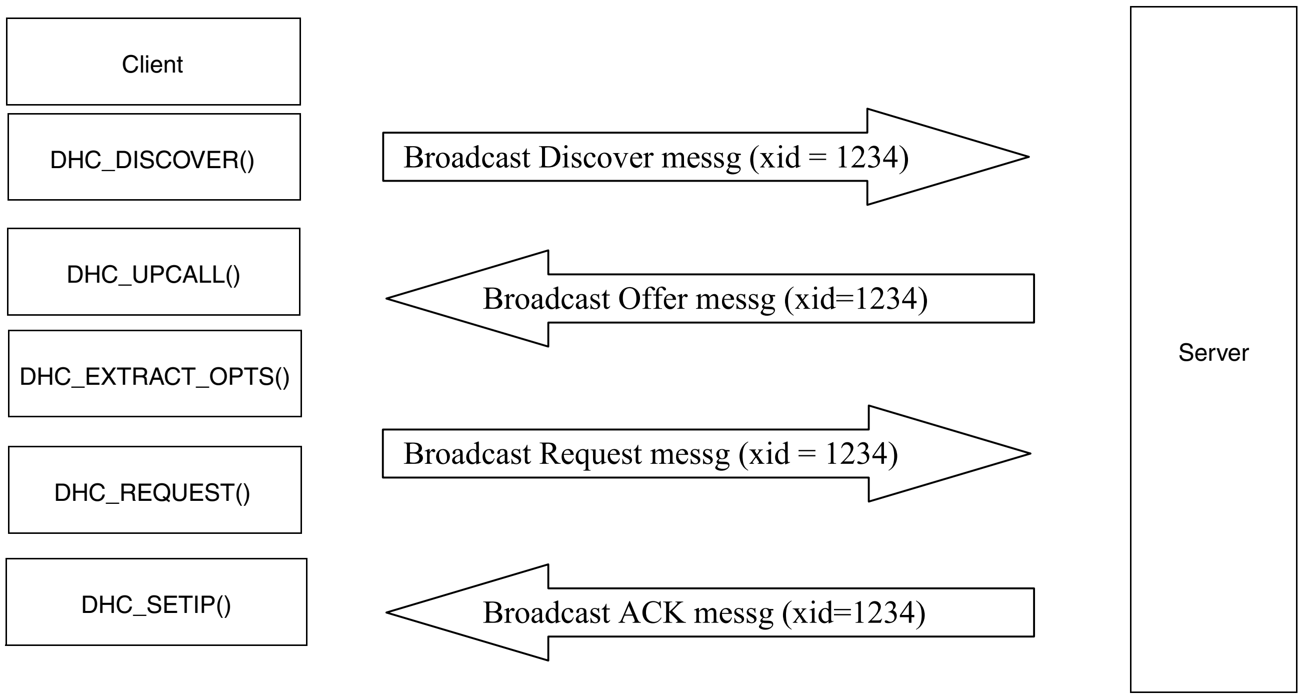

Client
DHC_DISCOVER()
DHC_UPCALL()
DHC_EXTRACT_OPTS()
DHC_REQUEST()
DHC_SETIP()
Broadcast Discover messg (xid = 1234)
Broadcast Offer messg (xid=1234)
Broadcast Request messg (xid = 1234)
Broadcast ACK messg (xid=1234)
Server

Figure 5. DHCP Transaction

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.

3.1 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.

4 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. Table 2 displays the DNS message.

Table 2. DNS Message

Question	The question for the name server
Answer	Resource record (RR) 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 to keep the DNS cache up to date.

The DNS client must be initialized by filling the dns_servers[] array with the IP addresses of DNS name servers. The dns_servers array[] is declared globally in dnsclnt.c. Any unused entries should be filled with zeros.

To use the DNS client, simply call an API function. The first time a name translation is requested, you must use the dns_query, dns_query_type, or gethostbyname functions. Each of these functions inserts the name and returned IP address into a cache. After the query is performed once, the dns_lookup() function can be used to get the information from the cache.

4.1 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.

4.2 DNS Usage Example

```
// url is a NULL terminated string containing the complete url
// The host name can be a dot notation ip address, or a Domain name
// example 75.18.69.29/index.html
//              or www.emgware.com/index.html
// The parse_ipad function returns a 0 if the string contains a ip address in dot notation
cp = parse_ipad(&ipaddr, &snbits, (char *)url);
// Is it a ip address?
if(cp)
{ // String is not a ip address
        // Pre-process string to eliminate any prefix (http://) and anything after domain name
        // Copy domain name only to allocated buffer;
        temp_buffer = (unsigned char *)npalloc(strlen((char *)url)+10);
        if(temp_buffer)
        {
                temp_buffer(0) = 0;        // in case

                // The preprocess_url function copies only the domain name into temp_buffer.
                (void)preprocess_url(temp_buffer, url);
                ipaddr = 0;

                // We will make multiple attempts at connecting to the DNS server
                for(i=0; i<EMG_HTTP_CLIENT_DNS_SEARCH_TIME_SECS; i++)
                {
                        tk_sleep(200);

                        // Send DNS client only the domain name (www.freescale.com)
                        test = gethostbyname((char *)temp_buffer);

                        if(test != NULL)
                        {
                                // IP address returned from DNS
                                ulp = (unsigned long *)test->h_addr_list[0];
                                ipaddr = *ulp;
                                break;
                        }
                }
                npfree(temp_buffer);

                if(!ipaddr)
                        return(EMG_HTTP_CLIENT_CONNECT_ERROR_DNS);
        }
        else
                return(EMG_HTTP_CLIENT_CONNECT_ERROR_ALLOC);
}
```

5 Advanced ColdFire TCP/IP Client Design—Zero Copy

The ColdFire TCP/IP stack supports two methods of client/server design. The first method uses a BSD sockets-like interface referred to as mini-IP. Client/server design using the mini-IP API is discussed in detail in *ColdFire TCP/UDP/IP Stack and RTOS*, AN3470. AN3470 also covers the complete TCP/IP stack and how to configure it.

This section covers designing a TCP/IP client using a minimum amount of RAM, while increasing performance. This is accomplished by using the zero-copy API, which gives you direct access to the TCP/IP stack.

The tradeoff for the performance and resource advantages is code complexity. Writing clients or servers using the mini-IP API is easier than using the zero-copy API, especially if you are familiar with BSD sockets.

5.1 ColdFire TCP/IP Zero-Copy API

`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 kicks 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)`

- Send 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 OK, 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 frees 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)`

- Frees netbuf pointed to by p.

```
M_SOCK m_socket
```

- Allocates a socket structure. The socket defaults to blocking. Returns a MSOCK structure if okay, null if error.

```
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.

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

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

```
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 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 by the m_ioctl function then m_connect returns EINPROGRESS. When the socket is flagged as non-blocking, the M_CALLBACK parameter is used to signal a completed connection, by calling the M_CALLBACK function.

  The m_connect function returns the error codes specified in the file msock.h. The MSOCK typedef is a pointer to a msocket structure:

  ```
  struct msocket
  {
          struct msocket * next;     // queue link
          unshort  lport;            // IP/port 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
  };
  ```

5.2 The Callback Function

The stack's TCP state machine contains hooks to call user functions under certain conditions. These hooks are referred to as callbacks. The callback function provides an asynchronous method of getting information from the stack, similar in concept to a interrupt.

The callback function is called directly from the TCP state machine, so it is important that the callback function contains no sleeps or long delays. The callback function should be treated as an interrupt service routine (ISR), although it is not always called under the context of a interrupt.

The application provides the address for the callback function in the call to m_connect. The address of the function is stored by the stack in the socket structure. The callback function declaration is shown below.

```
int wget_tcp_callback(int code, M_SOCK so, void * data)
```

When the callback function is called, it is passed a code identifying the reason it was called, a socket handle, and a data pointer. The data pointer is of type void, because it can point to different types of data depending on the code being reported. Not all codes provide data. The callback function is called under the conditions shown in Table 3.

Table 3. Callback Function Codes

Code	So	Data	Description
M_CLOSED	Handle	Null	Socket was closed.
M_OPENERR	Handle	Null	Not Used
M_OPENOK	Handle	Null	Connection to foreign host has been made.
M_TXDATA	Handle	Null	Data sent has been ACK'd by remote host.
M_RXDATA	Handle	Packet Structure	Indicates data received.

5.2.1 The Packet Structure

When a packet is received, the callback function is called with the void data * pointing to a PACKET structure. The PACKET structure is of type netbuf. Most of the data in the PACKET structure is not needed by your application. The two key elements are m_data and m_len.

- m_data—Points the actual user data in the packet.
- m_len—The length of the data.

```
typedef struct netbuf * PACKET;
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 (net) it came in on, 0-n
         ip_addr  fhost;      // IP address asociated 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
#ifdef MINI_TCP            // Mini TCP has no mbuf wrappers, so:
        char              * m_data;     // pointer to TCP data in nb_buff
        unsigned m_len;          // length of m_data
        struct netbuf * m_next; // sockbuf que link
#endif   /* MINI_TCP */

        struct ip_socopts *soxopts; // socket options
};
```

5.2.2 Example of Accessing Data from Packet in Callback Function

```
        PACKET pkt;                // pkt is a pointer to a netbuf structure

        // pkt->m_len = Number of Bytes Received
        // pkt->m_data = Data Buffer
        for(i=0; i<pkt->m_len; i++)
                ns_printf(lpio, %c, pkt->m_data[i]);
```

5.2.3 Callback Function Return Value

The callback function returns a value of zero or not zero. The stack ignores the callback's return function in every case except the M_RXDATA code. If the callback function returns a zero, the stack releases the packet immediately after the callback returns.

This is important because if the stack releases the packet immediately, the packet is no longer available to be read by a tcp_recv or a m_recv function. If the intent is to process the data directly in the callback, it returns zero. If the intent is to wake-up another task to process the data with a tcp_recv or m_recv, the callback must return a non-zero value.

5.2.4 Callback Function Example

```
//*****************************************************************************
// int wget_tcp_callback(int code, M_SOCK so, void * data)
//
// This function is called directly from the TCP state machine.
// The code is set based on the state the TCP stack was in when this function
// was called. Data is cast as void, because it points to different types
// of data depending on the code.
//
// M_CLOSED - Indicates that the socket was closed.
//                so = Socket handle
//                data points to NULL
//
// M_OPENERR - Not Used
//
// M_OPENOK - Indicates a connection to the foreign host has been made
//                so = Socket handle
//                data points to NULL
//
// M_TXDATA - Indicates Data sent by us has been ACK'd by remote host
//                so = Socket handle
//                data points to NULL
//
```

```
// M_RXDATA - Indicates data received
//                so = Socket handle
//                data points to PACKET structure
//                If 0 is returned, the stack frees the packet.
//                If !0 is returned, USER MUST FREE PACKET!!!!!
//**********************************************************************
int wget_tcp_callback(int code, M_SOCK so, void * data)
{
        PACKET pkt;
        int      i, k;
        int      e = 0;

        switch(code)
        {
                case M_OPENERR:// Not Used
                        break;

                case M_OPENOK:// Indicates a connection to the foreign host has been made
                        break;

                case M_CLOSED:// Indicates that the socket was closed
                        sclose = 1;
                        break;

                case M_RXDATA:          // Indicates data received
                        srx = EMG_HTTP_CLIENT_RX_TIME_SECS;
                        pkt = (PACKET)data;

                        // pkt->m_len = Number of Bytes Received
                        // pkt->m_data = Data Buffer
                        // We need to return 0, so stack releases packet
                        for(i=0; i<pkt->m_len; i++)
                        {
                                ns_printf(lpio, %c, pkt->m_data[i]);
                                k = emg_content_length_filter(pkt->m_data[i),
                                                          &filter_index,&filter_length);
                                if(k == 3) sclose = 1;
                                if(k ==2)
                                {
                                        if(filter_length)
                                        {
                                                filter_length--;
                                                if(filter_length == 0)
                                                        sclose = 1;
                                        }
                                }
                        }
                        break;

                case M_TXDATA:    // Indicates Data sent by us has been ACK'd by remote host
                        break;
                default:
                dtrap();             // not a legal case
                return 0;
        }
        USE_VOID(data);
        return e;
```

5.2.5 Connecting to a Remote Server Using the Zero-Copy API

The memory saving when using the zero-copy API is derived from the direct access to the packet buffers that the API provides. Normally, you would create the data in a buffer, then call the m_send function to send the data. The m_send function must then copy the data from the user buffer to the packet buffer. Almost twice the amount of RAM is required to perform the operation.

With the zero-copy API, you must allocate a packet buffer and insert your data directly into the packet. This eliminates the need for a user buffer when transmitting. The RAM advantage when receiving depends on the application. The bigger advantage of zero-copy when receiving is in the callback function. By getting a interrupt after a packet is received, the packet can be processed quickly, and the buffer freed as soon as possible. The faster packet buffers can be freed, the less that are needed. This results in an indirect savings in RAM and a direct increase in performance.

Steps required to connect to a remote server:

1. Create a socket by calling m_socket( ).

```
emg_tcp_communication_socket= m_socket( );
```

2. Initialize a sockaddr_in structure with a IP address and port.

```
// Init a socket structure with server Port Number
emg_tcp_sin.sin_addr.s_addr = ipaddr;
emg_tcp_sin.sin_port = port;
```

3. Call m_connect with the socket, a pointer to the sockaddr_in structure, and a pointer to the callback function.

```
// Socket is blocking.  The m_connect call blocks until it connects.
e = m_connect(emg_tcp_communication_socket, &emg_tcp_sin, cb);
if(e > 0)
{
     e = EMG_HTTP_CLIENT_CONNECT_ERROR_CONNECT;
     m_close(emg_tcp_communication_socket);
}
```

5.2.6 Sending Data Using the Zero-Copy API

The following steps are required to send data using the zero-copy API:

1. Allocate a packet buffer large enough for the data, using the tcp_pktalloc function.

```
for(i=0; i<EMG_HTTP_CLIENT_PACKET_WAIT_SEC; i++)
{
          pkt = tcp_pktalloc(llength);
          if(pkt)
                    break;
          tk_sleep(EMG_HTTP_CLIENT_TK_SLEEP_SEC);
}
```

2. Copy user data into the packet buffer

```
// Point buff to data section of packet
buff = (unsigned char *)pkt->m_data;

// Build GET request with url and extension, append the HTTP header
temp = emgstrcpy((unsigned char *)http_get_request, buff, &buff[llength]);
```

3. Set length of data in packet buffer

```
// Set length section of packet to (int)(temp-buff)
pkt->m_len = (unsigned int)(temp-buff);
```

4. Send data using tcp_send function

```
// Send data
emg = tcp_send(emg_tcp_communication_socket, pkt);   /* pass packet to tcp layer */
```

5. If tcp_send fails, you must free the packet.

```
// If tcp_send returns an error,  we need to release the packet.
if(emg)
         tcp_pktfree(pkt);
return(emg);
```

6 HTTP Client Firmware

The HTTP client is able to read web pages and XML data from the internet using a ColdFire processor. The HTTP client uses the DHCP client to automatically aquire a IP address and other TCP/IP information including the IP addresses of any DNS server. Then, the HTTP client uses the DNS client to translate any user-provided URLs to IP addresses.

The HTTP client uses the GET method to request a page from the server. Along with the GET request is the HTTP header. The HTTP header is hardcoded in the HTTP client by constant strings declared in the file emg_http_client.c.

6.1 The HTTP GET Request Header

The header is produced by combining the string constants:

```
const unsigned char http_get_request[] = “GET” ;
const unsigned char http_compatability[] = “HTTP/1.1\r\nAccept: */*\r\nAccept-Language:
         en-us\r\nUser-Agent: Mozilla/4.0\r\n”;
const unsigned char http_host[] = “Host:” ;
const unsigned char http_keep_alive[] = “Connection: Keep-Alive\r\n”;
const unsigned char http_done[] = “\r\n”;
```

`http_get_request[]`
— Followed by the filename portion of the requested URL (everything after /).

`http_compatability[]`
— The compatibility string tells the HTTP server the language we accept and the browser type we support.

`http_host[]`
— The host portion of the header is required by the HTTP specification. Most web servers fail if the host string is not in the header. The Apache server replies with a illegal header error. The host string must contain the name of the host server the client is connecting to. For example: www.emgware.com or www.freescale.com.

`http_keep_alive[]`
— The keep-alive string is only sent if the HTTP client wants to keep the TCP/IP connection alive after the file is received.

`http_done[]`

— The final line of the HTTP header is a blank line. Indicating to the server that there are no more fields.

6.2 HTTP Client Files

`emg_http_client.h`

- external function declarations and HTTP client constants.

`#define EMG_HTTP_CLIENT_TK_SLEEP_SEC 200`

- Number of cticks for a second (defined in main.c)

`#define EMG_HTTP_CLIENT_PACKET_WAIT_SEC10`

- Max number of seconds to wait for packet alloc to complete.

`#define EMG_HTTP_CLIENT_DNS_SEARCH_TIME_SECS 10`

- Max number of seconds to wait for DNS reply

`#define EMG_HTTP_CLIENT_RX_TIME_SECS 30`

- Max number of seconds to wait for RX data to terminate before closing.

`emg_http_client.c`

- The actual HTTP client.

`wget_byEricGregori.c`

- A command line driven example of how to use the http client. The wget command is added to the TCP/IP stack menu system. This allows you to get web pages directly from the serial port.

7 HTTP Client API

7.1 Function emg_HTTP_client_connect

```
int emg_HTTP_client_connect (unsigned char *url,
       unsigned short port, M_SOCK *shandle,
       M_CALLBACK(cb)
```

Where	Equals
url	Pointer to URL character string
port	Remote port number (80 for HTTP)
shandle	Socket handle populated by function
cb	Pointer to callback function
Returns	**If**
0	Success
Non-zero	Error. Codes defined in emg_http_client.h

The HTTP_client_connect function establishes a TCP/IP connection with the remote server. The URL string can contain an IP address in dot notation 75.18.69.29 or a domain name. The IP address or the domain name can be followed by a filename after a /. There is no hard limit on url sizes, although the GET header with URL must fit in a single packet (1536 bytes).

Sample URLs:

- www.freescale.com
- 75.18.69.29
- www.emgware.com/appnotes.htm
- 75.18.69.29/appnotes.htm

7.2 Function emg_HTTP_client_close

```
void emg_HTTP_client_close (
      M_SOCK emg_tcp_communication_socket)
```

Where	Equals
M_SOCK	Socket handle returned by connect

7.3 Function emg_HTTP_client_get

```
int emg_HTTP_client_get (unsigned char *url,
      unsigned char *extension,
      M_SOCK emg_tcp_communication_socket,
      unsigned char keepalive)
```

Where	Equals
url	Pointer to URL character string (same as connect)
extension	Pointer to a string that is added to the end of the url
M_SOCK	Socket handle returned by connect
keepalive	0 to close connection after receiving file 1 to keep connection alive after receiving file
Returns	**If**
0	Success

The HTTP_client_get function sends the GET request followed by the HTTP header to the remote server. The data returned by the server is sent to the user application via the callback function pointed to in the connect call. Data is sent to the callback one packet at a time, so a typical web page actually gets many data callbacks.

The extension string is for search strings or any other type of changing data in the URL. This allows you to specify a constant url and only change the dynamic portion at the end of the URL (for example, search parameters for a search engine).

The keep-alive flag is zero for normal operation. When it is zero, the HTTP client does not send the keep-alive request in the HTTP header. This allows the server and the client to close gracefully after a file is transferred. If the keep-alive flag is set, the keep-alive request is sent to the server, and it does not close after the file is sent. This allows another file to be downloaded without calling the connect function again.

7.4 Function emg_content_length_filter

`int emg_content_length_filter(unsigned char data, unsigned int *index, unsigned int *length)`	
Where	**Equals**
data	1 byte of data from packet
index	Pointer to unsigned integer. Set variable (not pointer) to zero before calling filter the first time
length	Pointer to number of bytes to download after header. Set variable (not pointer) to zero before calling filter the first time.
Returns	**If**
0	Nothing found yet
1	Length found
2	Terminator found
3	Zero length found

The content_length_filter function is a state machine that filters out the content length from a HTTP header. As the user application receives packets from the server, this function is called for each byte in the packet (until a two or three is returned). The length variable is populated if a content-length: field is found in the HTTP header. The length can determine when all the data has been received. No action has to take place if a one is returned, the length variable is not corrupted.

To filter out the header, send packet bytes to filter while your application ignores these bytes, until the filter returns a two or three. The filter always returns a two or three, assuming a valid (correctly terminated) HTTP header. Until the filter sees the terminator, it returns zero (or one after the length variable is populated).

The index variable is a state variable used by the filter. Your application should not alter the index variable, other than to initialize it to zero before calling the filter for the first time. The length variable should also be set to zero before calling the filter the first time.

8 Wget Command — Example of Using the HTTP Client

The wget command is a command often found in Linux distributions that transfers files using the HTTP protocol. The wget command is a console based HTTP client. Using the menuing system provided by the ColdFire TCP/IP stack (explained in AN3470) and the HTTP client, wget functionality can be added to the ColdFire TCP/IP stack.

command:

```
wget url
```

Where url is a string containing the domain name (or ip address in dot notation) followed by a '/' and a filename (or any other legal url parameters).

Example url strings:

- The link to download the base software for this application note:
  - http://www.freescale.com/webapp/sps/download/license.jsp?colCode=AN3470SW&location=null&fpsp=1
- The link to AN3470
  - http://www.freescale.com/files/microcontrollers/doc/app_note/AN3470.pdf?fpsp=1

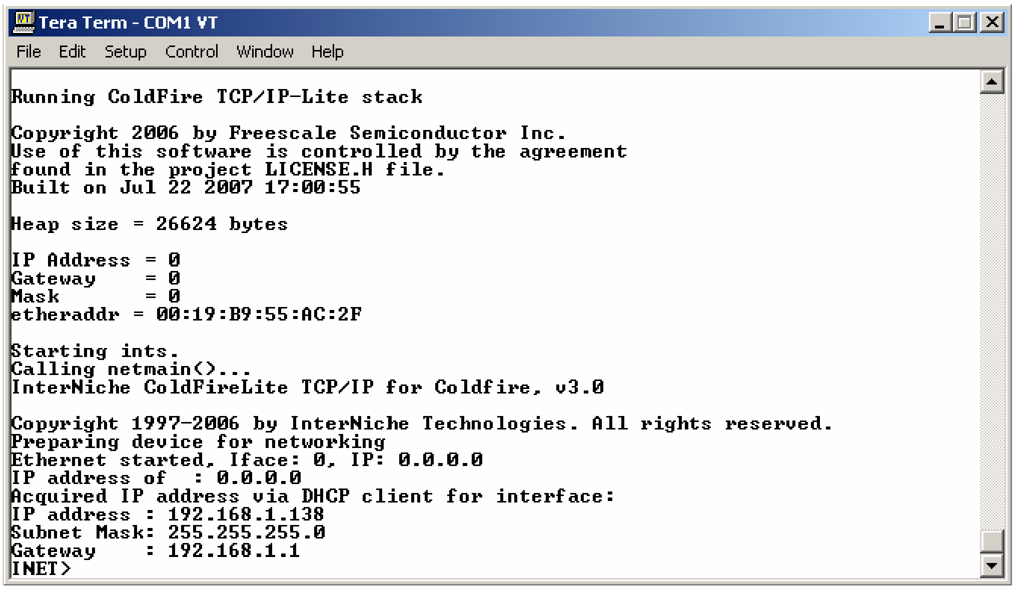
NOTE

The standard menu system command buffer is limited to 32 characters. To increase the menu systems character buffer, make the following changes:

In the file iuart.c set UART_RXBUFSIZE to whatever size you want the command buffer to be.

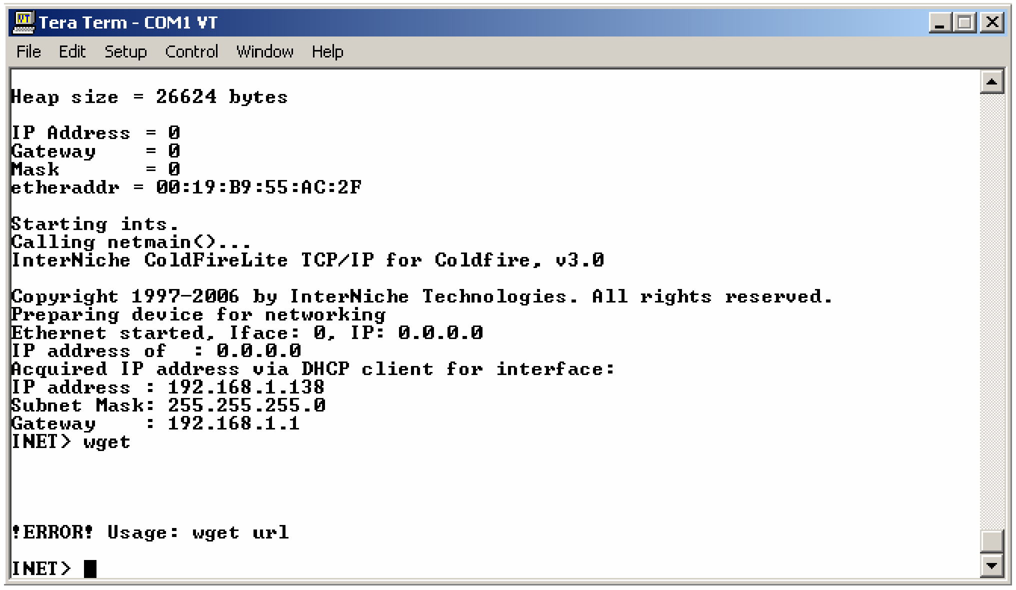
```
#ifndef UART_RXBUFSIZE
#define UART_RXBUFSIZE       256// command buffer is 256 bytes
#endif
```

8.1 wget Usage Examples



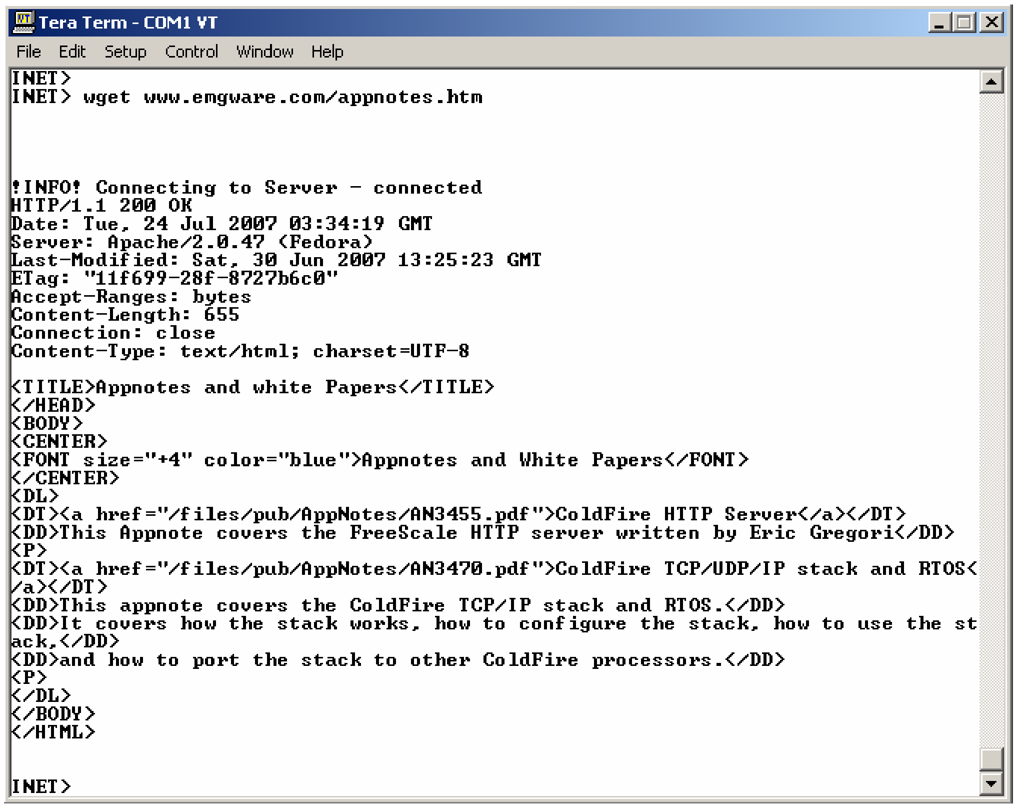
```
Running ColdFire TCP/IP-Lite stack

Copyright 2006 by Freescale Semiconductor Inc.
Use of this software is controlled by the agreement
found in the project LICENSE.H file.
Built on Jul 22 2007 17:00:55

Heap size = 26624 bytes

IP Address = 0
Gateway    = 0
Mask       = 0
etheraddr = 00:19:B9:55:AC:2F

Starting ints.
Calling netmain()...
InterNiche ColdFireLite TCP/IP for Coldfire, v3.0

Copyright 1997-2006 by InterNiche Technologies. All rights reserved.
Preparing device for networking
Ethernet started, Iface: 0, IP: 0.0.0.0
IP address of   : 0.0.0.0
Acquired IP address via DHCP client for interface:
IP address : 192.168.1.138
Subnet Mask: 255.255.255.0
Gateway    : 192.168.1.1
INET>
```

Figure 6. Stack Bootup and Command Prompt



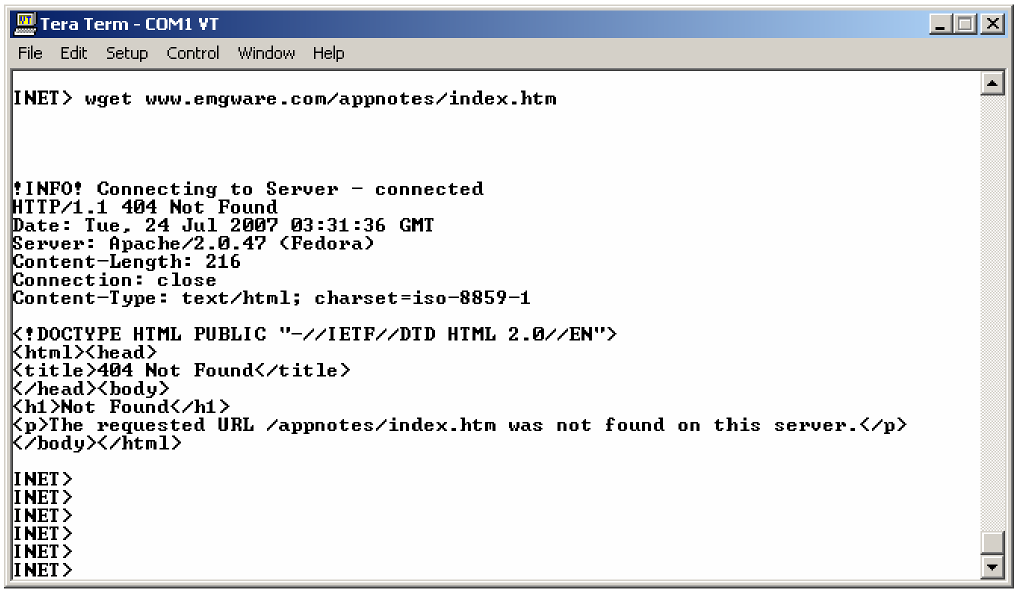
```
Heap size = 26624 bytes

IP Address  = 0
Gateway     = 0
Mask        = 0
etheraddr = 00:19:B9:55:AC:2F

Starting ints.
Calling netmain()...
InterNiche ColdFireLite TCP/IP for Coldfire, v3.0

Copyright 1997-2006 by InterNiche Technologies. All rights reserved.
Preparing device for networking
Ethernet started, Iface: 0, IP: 0.0.0.0
IP address of  : 0.0.0.0
Acquired IP address via DHCP client for interface:
IP address : 192.168.1.138
Subnet Mask: 255.255.255.0
Gateway    : 192.168.1.1
INET> wget



!ERROR! Usage: wget url

INET> █
```

Figure 7. wget Usage



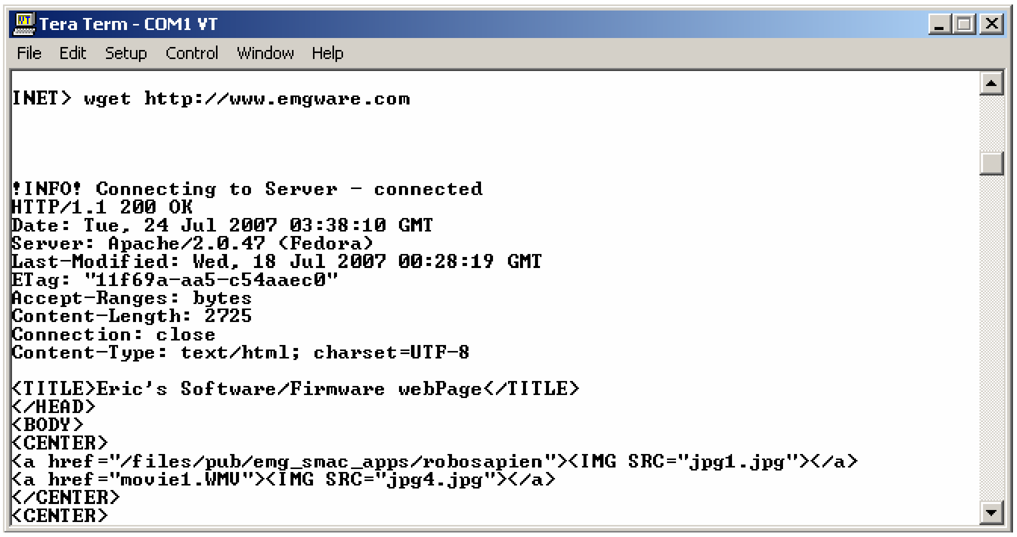
```
INET>
INET> wget www.emgware.com/appnotes.htm




!INFO! Connecting to Server - connected
HTTP/1.1 200 OK
Date: Tue, 24 Jul 2007 03:34:19 GMT
Server: Apache/2.0.47 (Fedora)
Last-Modified: Sat, 30 Jun 2007 13:25:23 GMT
ETag: "11f699-28f-8727b6c0"
Accept-Ranges: bytes
Content-Length: 655
Connection: close
Content-Type: text/html; charset=UTF-8

<TITLE>Appnotes and white Papers</TITLE>
</HEAD>
<BODY>
<CENTER>
<FONT size="+4" color="blue">Appnotes and White Papers</FONT>
</CENTER>
<DL>
<DT><a href="/files/pub/AppNotes/AN3455.pdf">ColdFire HTTP Server</a></DT>
<DD>This Appnote covers the FreeScale HTTP server written by Eric Gregori</DD>
<P>
<DT><a href="/files/pub/AppNotes/AN3470.pdf">ColdFire TCP/UDP/IP stack and RTOS<
/a></DT>
<DD>This appnote covers the ColdFire TCP/IP stack and RTOS.</DD>
<DD>It covers how the stack works, how to configure the stack, how to use the st
ack.</DD>
<DD>and how to port the stack to other ColdFire processors.</DD>
<P>
</DL>
</BODY>
</HTML>


INET>
```

Figure 8. Successful File Transfer



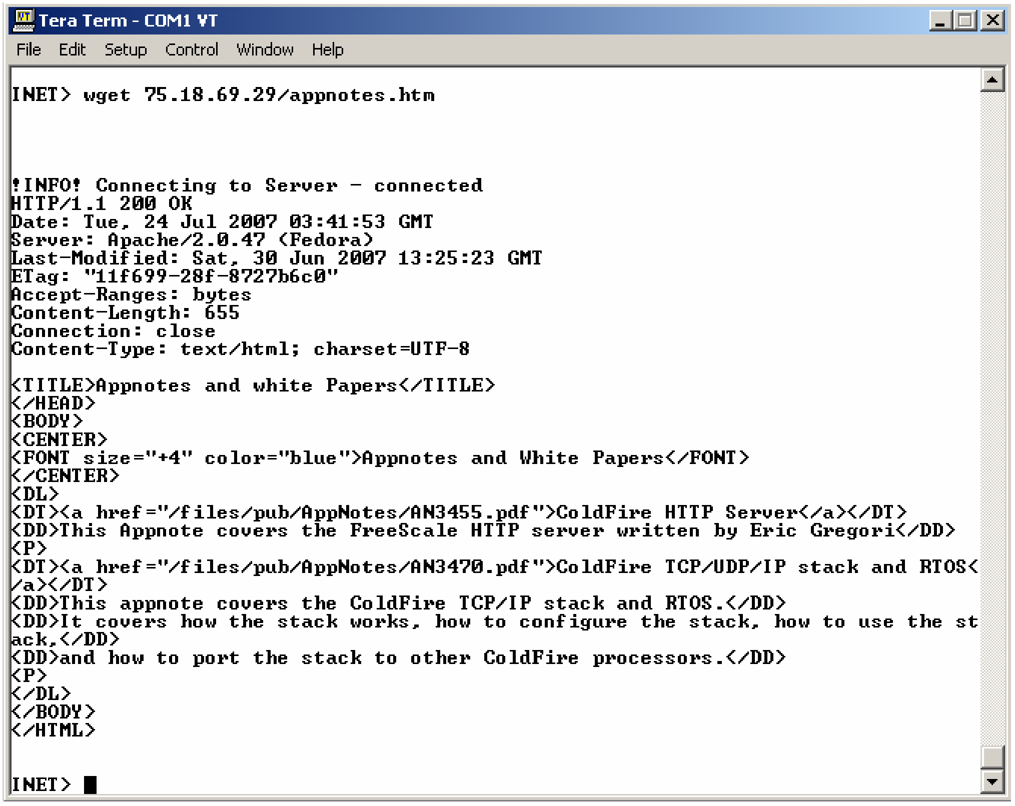
```
INET> wget www.emgware.com/appnotes/index.htm



!INFO! Connecting to Server - connected
HTTP/1.1 404 Not Found
Date: Tue, 24 Jul 2007 03:31:36 GMT
Server: Apache/2.0.47 (Fedora)
Content-Length: 216
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /appnotes/index.htm was not found on this server.</p>
</body></html>

INET>
INET>
INET>
INET>
INET>
INET>
```

Figure 9. 404 File Not Found Example



```
INET> wget http://www.emgware.com



!INFO! Connecting to Server - connected
HTTP/1.1 200 OK
Date: Tue, 24 Jul 2007 03:38:10 GMT
Server: Apache/2.0.47 (Fedora)
Last-Modified: Wed, 18 Jul 2007 00:28:19 GMT
ETag: "11f69a-aa5-c54aaec0"
Accept-Ranges: bytes
Content-Length: 2725
Connection: close
Content-Type: text/html; charset=UTF-8

<TITLE>Eric's Software/Firmware webPage</TITLE>
</HEAD>
<BODY>
<CENTER>
<a href="/files/pub/emg_smac_apps/robosapien"><IMG SRC="jpg1.jpg"></a>
<a href="movie1.WMV"><IMG SRC="jpg4.jpg"></a>
</CENTER>
<CENTER>
```

Figure 10. URL With Protocol Type



```
INET> wget 75.18.69.29/appnotes.htm



!INFO! Connecting to Server - connected
HTTP/1.1 200 OK
Date: Tue, 24 Jul 2007 03:41:53 GMT
Server: Apache/2.0.47 (Fedora)
Last-Modified: Sat, 30 Jun 2007 13:25:23 GMT
ETag: "11f699-28f-8727b6c0"
Accept-Ranges: bytes
Content-Length: 655
Connection: close
Content-Type: text/html; charset=UTF-8

<TITLE>Appnotes and white Papers</TITLE>
</HEAD>
<BODY>
<CENTER>
<FONT size="+4" color="blue">Appnotes and White Papers</FONT>
</CENTER>
<DL>
<DT><a href="/files/pub/AppNotes/AN3455.pdf">ColdFire HTTP Server</a></DT>
<DD>This Appnote covers the FreeScale HTTP server written by Eric Gregori</DD>
<P>
<DT><a href="/files/pub/AppNotes/AN3470.pdf">ColdFire TCP/UDP/IP stack and RTOS<
/a></DT>
<DD>This appnote covers the ColdFire TCP/IP stack and RTOS.</DD>
<DD>It covers how the stack works, how to configure the stack, how to use the st
ack,</DD>
<DD>and how to port the stack to other ColdFire processors.</DD>
<P>
</DL>
</BODY>
</HTML>

INET> █
```

Figure 11. Using a Dot Notation IP Address

9 Really Simple Syndication (RSS)

RSS feeds are available everywhere on the internet. They convey information such as local weather, search engine results, DVD queue information, DVD new releases, headline news, and sports news. The idea behind the RSS feed is to deliver textual, dynamic information in a standard simple format.

RSS originated in 1999, with the idea to provide content in a simple, easy-to-understand format. Instead of describing a document in HTML, RSS feeds use XML to describe data. A RSS feed is simply a XML document containing data. The methods used to convey the data within the XML document are described in the RSS 2.0 specification. All RSS files must conform to the XML 1.0 specification. RSS feeds generally use HTTP as the transport mechanism.

9.1 Extensible Markup Language (XML)

XML is a language that describes and parses information and is similar to structures in C. Data is organized into elements, which are surrounded by a start and end tag. End tag names must be the same as the start tag names. End tags are identified by the addition of a / before the tag name. The name in the start and end tags gives the element's type.

The XML 1.0 specification can be found at http://www.w3.org/TR/REC-xml

9.1.1 Tags

An example tag is as shown below.

```
<TITLE>Advanced ColdFire TCP/IP Clients</TITLE>
```

TITLE is the element type, <TITLE> is the start tag, and </TITLE> is the end tag. The data is between the tags. Like a C data structure, the data is associated with the type. The data between the start and end tags is referred to as the element's content.

Elements can contain other elements, providing a method of grouping data of different type under a single name. Like a C structure, the particular piece of data is referenced by specifying a path to the data. For example:

```
<APPNOTES>
    <BYEG>
        <AN3455>ColdFire HTTP server</AN3455>
        <AN3470>ColdFire TCP/IP Stack</AN3470>
        <AN3492>ColdFire USB Stack</AN3492>
    </BYEG>
</APPNOTES>
```

9.1.2 Special Characters and Escape Sequences

The & , <, and > characters are special XML characters. These characters define XML tags and escape sequences. Escape sequences specifiy a character with a code as opposed to a single symbol. Escape sequences start with an ampersand and end with a semicolon. To use these characters as an element’s content, use the following substitutions:

Table 4. Special Characters

Character	Escape Sequence
<	&lt;
>	&gt;
&	&amp;

9.1.3 CDATA Sections

The CDATA section is the exception to the rule in XML. Any text specified in a CDATA section is ignored by the XML parser, allowing the use of special characters without being escaped. A CDATA section starts with a <!CDATA[string, and is terminated with a))> string. Anything between the brackets is ignored by XML. For example:

```
<![CDATA[ character data here is ignored by XML parser ))>
<![CDATA[ <THIS_IS_NOT_A_TAG> &lt; ))>
```

Because the text between the brackets is ignored by the XML parser, the tag and the escape sequence are ignored, and are interpreted as text or character data.

9.1.4 Finding the Text or Character Data in a XML Document

From the XML 1.0 specification, all text that is not markup constitutes the character data of the document. That includes all the text between the brackets in a CDATA section, and any text not between < > brackets in the main body. Escape sequences in the main body represent a single piece of character data.

To filter the character data from a XML document, remove all the tags and < > brackets. Then, translate the escape sequences into actual characters.

9.1.5 Sample XML File

In the sample XML document below, data is encapsulated by tags, which describe the data. It resembles a C structure. To find the desired data in a XML document, look for the start and end tags with the name of the type of data you want. The actual data associated with the desired type is between the tags.

From: http://www.weather.gov/data/current_obs/KUGN.xml

```
<?xml version=1.0 encoding=ISO-8859-1 ?>
<current_observation version=1.0 xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:noNamespaceSchemaLocation=http://www.weather.gov/data/current_obs/current_observation.xsd
>
<credit>NOAA's National Weather Service</credit>
<credit_URL>http://weather.gov/</credit_URL>
<image>
<url>http://weather.gov/images/xml_logo.gif</url>
<title>NOAA's National Weather Service</title>
<link>http://weather.gov</link>
</image>
<suggested_pickup>15 minutes after the hour</suggested_pickup>
<suggested_pickup_period>60</suggested_pickup_period>
<location>Chicago / Waukegan, Waukegan Regional Airport, IL</location>
<station_id>KUGN</station_id>
<latitude>42.420</latitude>
<longitude>-87.870</longitude>
<observation_time>Last Updated on Jul 28, 10:52 am CDT</observation_time>
<observation_time_rfc822>Sat, 28 Jul 2007 10:52:00 -0500 CDT</observation_time_rfc822>
<weather>Overcast</weather>
<temperature_string>73 F (23 C)</temperature_string>
<temp_f>73</temp_f>
<temp_c>23</temp_c>
<relative_humidity>81</relative_humidity>
<wind_string>From the Northeast at 13 Gusting to 21 MPH</wind_string>
<wind_dir>Northeast</wind_dir>
<wind_degrees>30</wind_degrees>
<wind_mph>12.65</wind_mph>
<wind_gust_mph>21</wind_gust_mph>
<pressure_string>29.98 (1014.1 mb)</pressure_string>
<pressure_mb>1014.1</pressure_mb>
<pressure_in>29.98</pressure_in>
<dewpoint_string>67 F (19 C)</dewpoint_string>
```

```
<dewpoint_f>67</dewpoint_f>
<dewpoint_c>19</dewpoint_c>
<heat_index_string>73 F (23 C)</heat_index_string>
<heat_index_f>73</heat_index_f>
<heat_index_c>23</heat_index_c>
<windchill_string>NA</windchill_string>
<windchill_f>NA</windchill_f>
<windchill_c>NA</windchill_c>
<visibility_mi>10.00</visibility_mi>
<icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base>
<icon_url_name>ovc.jpg</icon_url_name>
<two_day_history_url>http://www.weather.gov/data/obhistory/KUGN.html</two_day_history_url>
<ob_url>http://www.nws.noaa.gov/data/METAR/KUGN.1.txt</ob_url>
<disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
<copyright_url>http://weather.gov/disclaimer.html</copyright_url>
<privacy_policy_url>http://weather.gov/notice.html</privacy_policy_url>
</current_observation>
```

9.1.6 Problem with XML Documents

The problem with XML documents is the flexibility of the tag names. The creator of the XML document can use any name to describe the data. A standard is required to standardize the tag names and how data is associated with type. That standard is the RSS standard.

9.2 RSS Specification

RSS is a dialect of XML. RSS embeds HTML constructs into the XML architecture. RSS also defines a set group of elements, and a general template using those elements. There are many elements defined in the specification. This document concentrates on the elements of interest for the RSS appliance.

A typical RSS file is shown below

```
<channel>
      <title>The name of the channel</title>
      <link>URL to the HTML website corresponding to this channel</link>
      <description>Text describing the channel</description>
      <item>
          < title>Item1 Title</title>
          <link>URL of item 1</link>
          <description>Text for item 1</description>
      </item>
      <item>
          <title>Item2 Title</title>
          <link>URL of item 2</link>
          <description>Text for item 2</description>
      </item>
</channel>
```

The organization of the RSS file can be compared to a newspaper. The channel is the name of the paper, the items are the articles in the paper, the titles are the article titles, and the descriptions are the article’s text.

9.2.1 Sample RSS File

The sample RSS file is the same data as the sample XML file above. The RSS standard defines the title and description tag names. Those tags contain the data needed. All RSS 2.0 compliant feeds contain title and description tags.

From: http://www.weather.gov/data/current_obs/KUGN.rss

```
<?xml version=1.0 encoding=ISO-8859-1 ?>
<rss version=2.0 xmlns:dc=http://purl.org/dc/elements/1.1/>
<channel>
<title>Weather at Chicago / Waukegan, Waukegan Regional Airport, IL - via NOAA's National
Weather Service</title>
<link>http://www.weather.gov/data/current_obs/</link>
<lastBuildDate>Sat, 28 Jul 2007 16:32:11 UT</lastBuildDate>
<ttl>60</ttl>
<description>Weather conditions from NOAA's National Weather Service.</description>
<language>en-us</language>
<managingEditor>robert.bunge@noaa.gov</managingEditor>
<webMaster>w-nws.webmaster@noaa.gov</webMaster>
<image>
<url>http://www.weather.gov/images/xml_logo.gif</url>
<title>NOAA - National Weather Service</title>
<link>http://www.weather.gov/data/current_obs/</link>
</image>
<item>
<title>Overcast and 73 degrees F at Chicago / Waukegan, Waukegan Regional Airport, IL</title>
<link>http://weather.noaa.gov/weather/current/KUGN.html</link>
<description>
<![CDATA[
<img src=http://weather.gov/weather/images/fcicons/ovc.jpg class=noaaWeatherIcon width=55
height=58 alt=Overcast style=float:left; /><br />
 ))>
 Winds are Northeast at 13 Gusting to 21 MPH. The pressure is 29.98 (1014.1 mb) and the humidity
is 81%. The heat index is 73. Last Updated on Jul 28, 10:52 am CDT.
</description>
<guid isPermaLink=false>Sat, 28 Jul 2007 10:52:00 -0500 CDT</guid>
</item>
</channel>
</rss>
```

9.3 RSS/XML Character Data Filter

To extract the character data (the information you actually want to read) from the RSS stream, all the metatext must be filtered out. Any HTML that can be processed must be translated and processed. For instance, a <br> is HTML for a new line. This would appear as &lt;br&gt; in the RSS stream. The filter must correctly translate &lt;br&gt; into a carriage return and line feed. Other HTML tags that are routinely embedded in character data include paragraph tags <p> and image tags <IMG …>. In the stream these tags appear as &lt;p&gt; and &lt;IMG …&gt; repectively. The paragraph tab can be translated to a carriage return and line feed, but the image tag must be ignored unless the embedded system can process images.

9.3.1 RSS/XML Character Data Filter State Machine

The character data filter is implemented as a finite state machine. The state machine uses only two global variables, a state variable, and a FILO. The purpose of the FILO is to store previous characters. Each byte entered into the filter is shifted left into the FILO. The most recent character is at location filo_buff[FILO_BUFF_SIZE-1). The FILO buffer size must be larger then the largest tag name expected. The FILO buffer size is set with the FILO_BUFF_SIZE macro.

```
#define FILO_BUFF_SIZE 32
```

States:

```
      STATE_ZERO
      STATE_TAGSEARCH
      STATE_INTAG
      STATE_PRINT
      STATE_SKIP
      STATE_SKIP_NON_ASCII
      STATE_SKIP_SPACE
      STATE_CDATA
      STATE_CDATA_PRINT
      STATE_CDATA_SKIP_AMP
      STATE_SKIP_INTAG
      STATE_IN_AMPERSAND
```

Global Variables:

```
      unsigned char filo_buff[FILO_BUFF_SIZE);
      unsigned char state;
unsigned char EMG_rss_text_filter(unsigned char data, unsigned char **tag_filter)
```

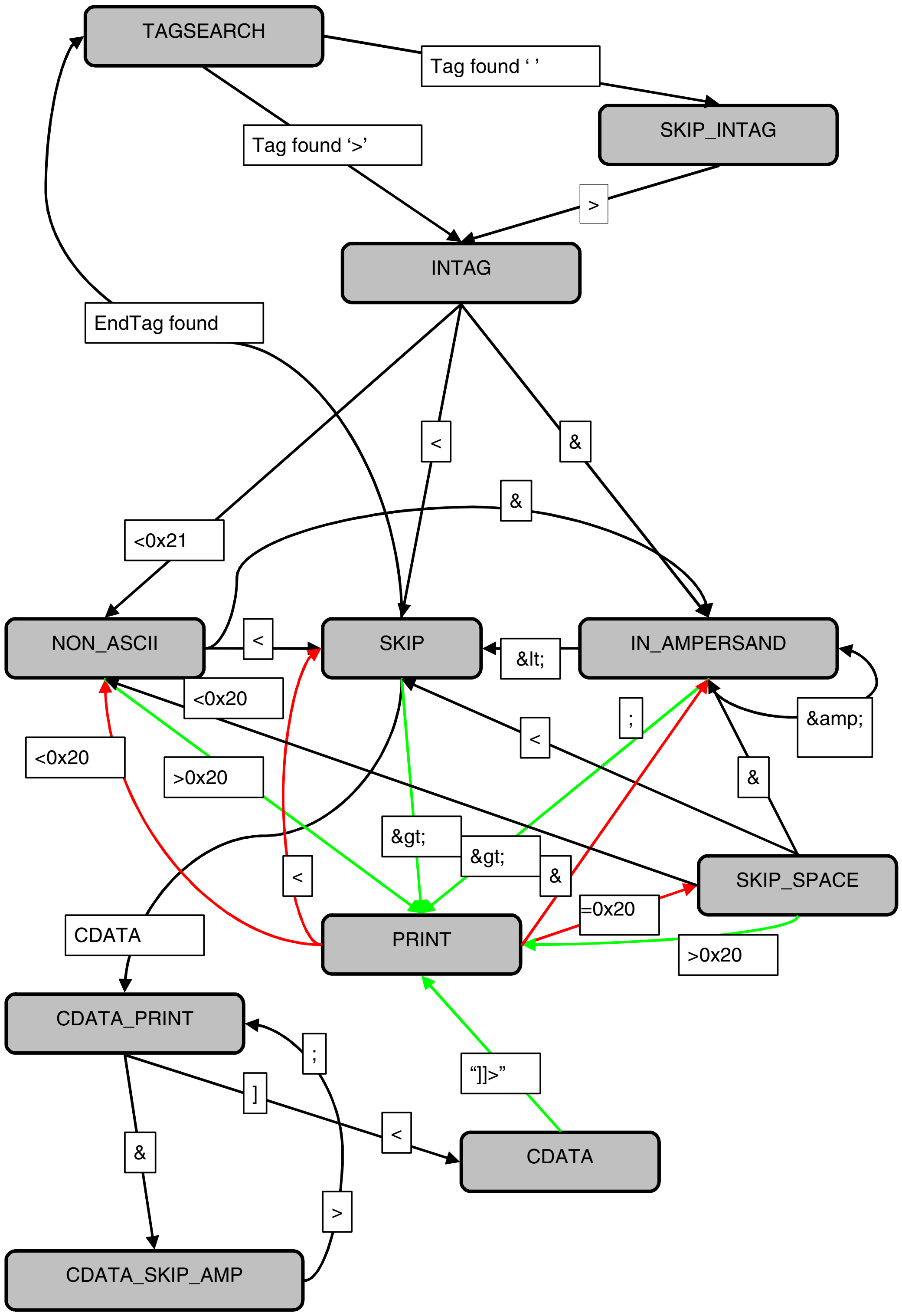


Figure 12. Character Data Filter State Machine

The filter takes in a XML or RSS data stream, and a list of tags. It outputs the character data only from the selected tags. The tag list is a array of pointers to the tag strings the filter should be filtering character data from.

Normally, the filter returns zero. When the filter processes the > in a tag in the list, the filter returns the index into the filter array for the tag that it found plus one. For example, after detecting a <title> tag in a RSS or XML stream the filter returns one. After detecting a <description> tag in a stream, the filter returns two.

```
Const unsigned char *tag_filter[] =
      {
      {(const unsigned char *)”title”},
      {(const unsigned char *)”description”},
      {(const unsigned char *)””}
      };
```

Sample tag_filter pointer array

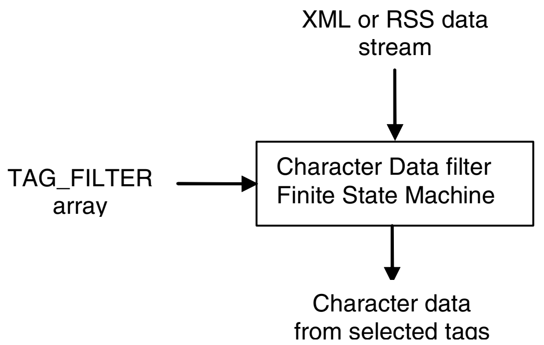

XML or RSS data stream
TAG_FILTER array
Character Data filter Finite State Machine
Character data from selected tags

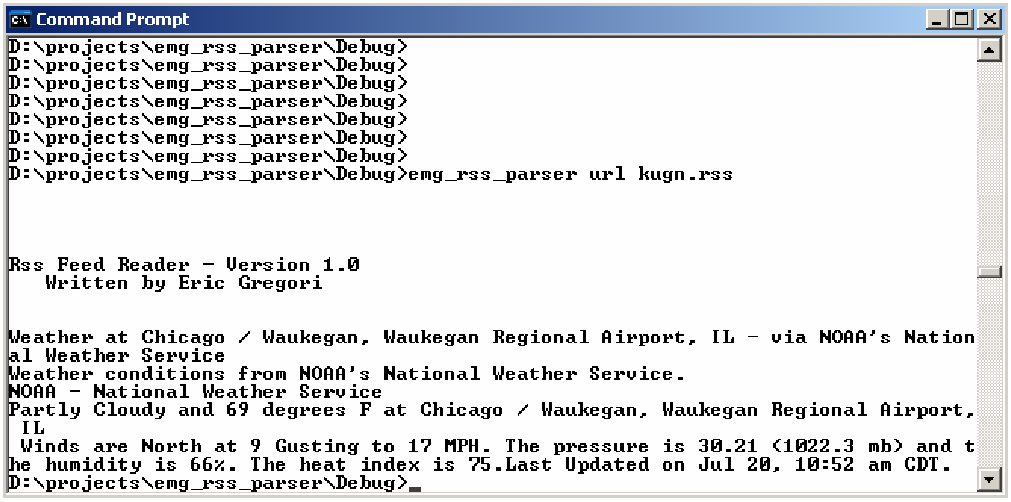
Figure 13. Character Data Filter Usage

9.3.2 Character Data Filter Excercisor PC Application

A PC application is available to assist in verifiying the state machine. The PC application opens the XML or RSS stream as a file. The stream is passed through the finite state machine filter, and the result is sent to the console.

```
// For testing read data from file, and send to filter
EMG_init_rss_text_filter;
for(; !feof(input_file);)
{
        if ((length = fread(temp, 1, PACKET_SIZE, input_file))==0)
                break;
        if (ferror(input_file))
                break;
        for(i=0; i<length; i++)
        {
                ret = EMG_rss_text_filter(temp[i), (unsigned char **)tag_filter);
                if(ret) // Print a new line after each tag
                        printf(\n);
        }
}
```

Using the RSS streams from Section 9.2.1, “Sample RSS File”, the filtered results using the PC excercisor (filtereing <title> and <description> tags) are shown below.



```
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>
D:\projects\emg_rss_parser\Debug>emg_rss_parser url kugn.rss



Rss Feed Reader - Version 1.0
   Written by Eric Gregori


Weather at Chicago / Waukegan, Waukegan Regional Airport, IL - via NOAA's Nation
al Weather Service
Weather conditions from NOAA's National Weather Service.
NOAA - National Weather Service
Partly Cloudy and 69 degrees F at Chicago / Waukegan, Waukegan Regional Airport,
 IL
 Winds are North at 9 Gusting to 17 MPH. The pressure is 30.21 (1022.3 mb) and t
he humidity is 66%. The heat index is 75.Last Updated on Jul 20, 10:52 am CDT.
D:\projects\emg_rss_parser\Debug>_
```

Figure 14. Filter Results for Kugn.rss Feed

9.3.3 Modified wget Command

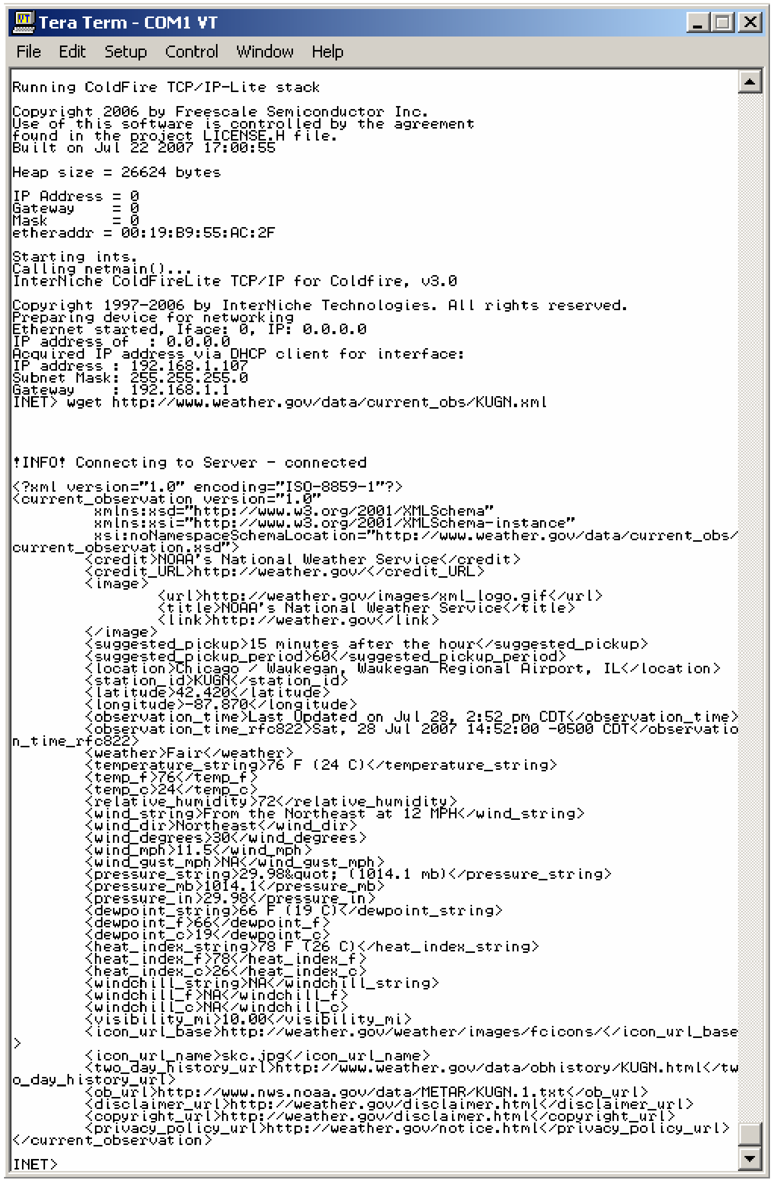
The wget command uses the HTTP protocol to get a file from a remote server. In this application, the wget command has been upgraded to support the character data filter. The new syntax for the wget command is:

```
wget <tag1><tag2> url
```

where <tag1> and <tag2> are optional and must be surrounded by <>.

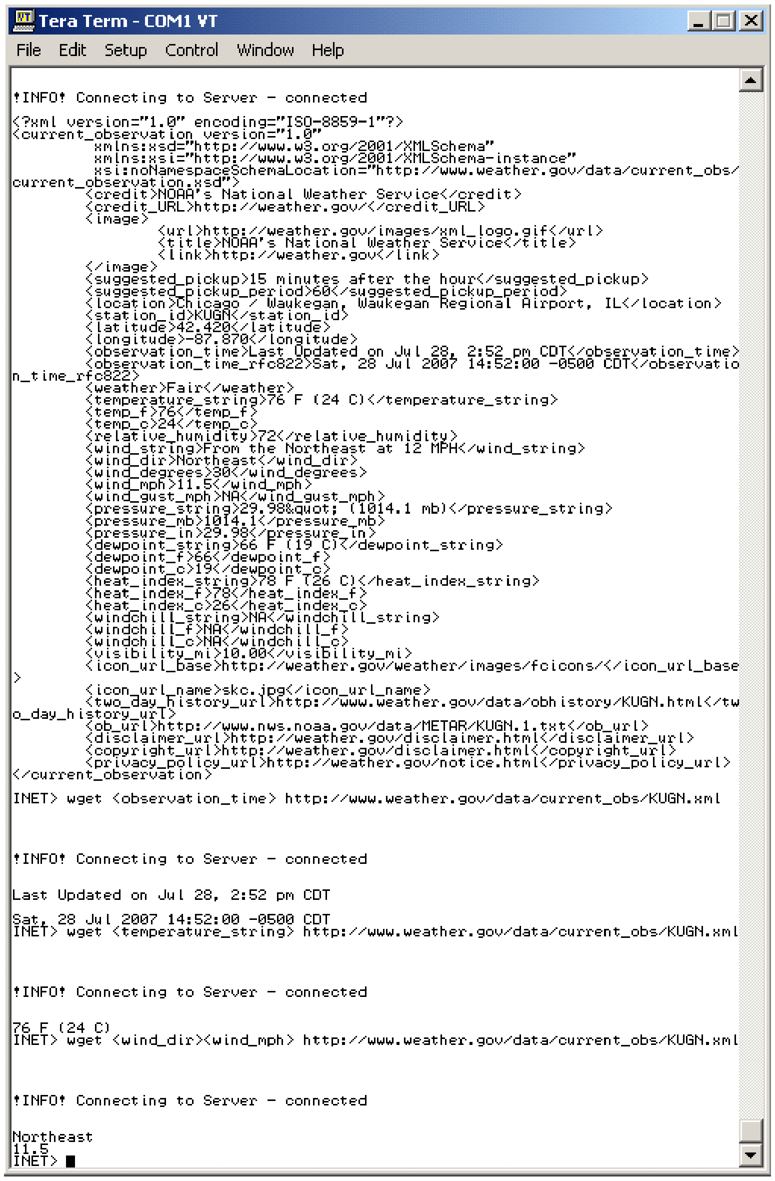
Also, there is no space between tags. The number of accepted tags can be increased using the following macro:

```
#define TAG_FILTER_SIZE 3
```



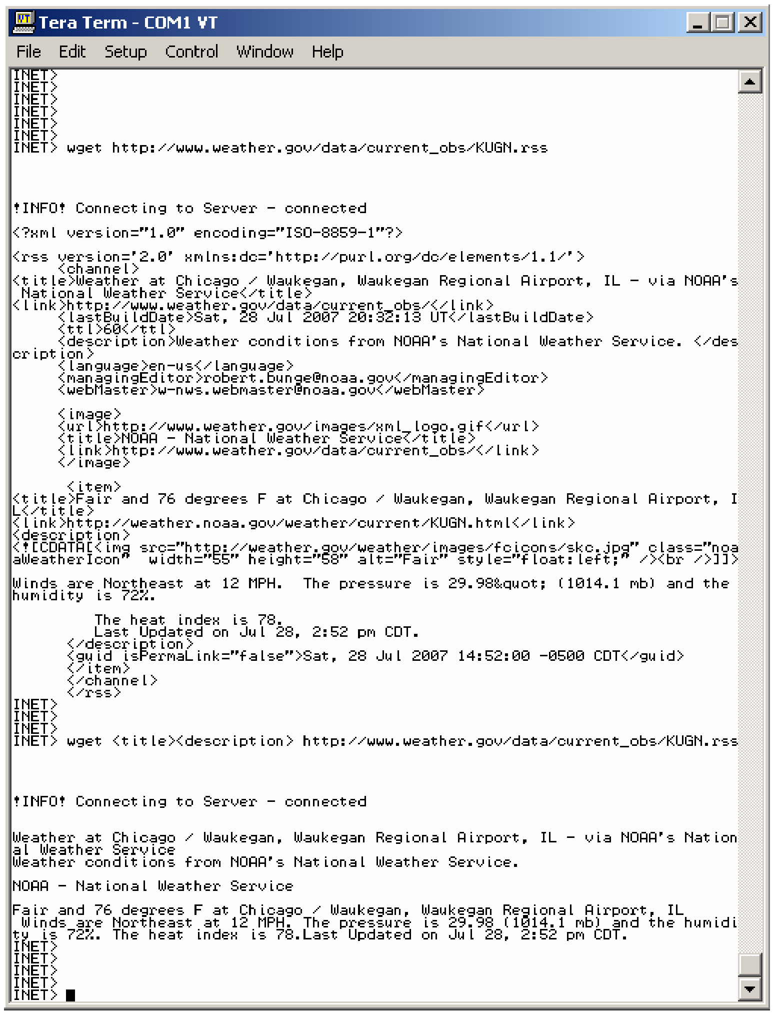
```
Running ColdFire TCP/IP-Lite stack

Copyright 2006 by Freescale Semiconductor Inc.
Use of this software is controlled by the agreement
found in the project LICENSE.H file.
Built on Jul 22 2007 17:00:55

Heap size = 26624 bytes

IP Address = 0
Gateway    = 0
Mask       = 0
etheraddr = 00:19:B9:55:AC:2F

Starting ints.
Calling netmain()...
InterNiche ColdFireLite TCP/IP for Coldfire, v3.0

Copyright 1997-2006 by InterNiche Technologies. All rights reserved.
Preparing device for networking
Ethernet started, Iface: 0, IP: 0.0.0.0
IP address of  : 0.0.0.0
Acquired IP address via DHCP client for interface:
IP address : 192.168.1.107
Subnet Mask: 255.255.255.0
Gateway    : 192.168.1.1
INET> wget http://www.weather.gov/data/current_obs/KUGN.xml



!INFO! Connecting to Server - connected

<?xml version="1.0" encoding="ISO-8859-1"?>
<current_observation version="1.0"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.weather.gov/data/current_obs/
current_observation.xsd">
        <credit>NOAA's National Weather Service</credit>
        <credit_URL>http://weather.gov/</credit_URL>
        <image>
                <url>http://weather.gov/images/xml_logo.gif</url>
                <title>NOAA's National Weather Service</title>
                <link>http://weather.gov</link>
        </image>
        <suggested_pickup>15 minutes after the hour</suggested_pickup>
        <suggested_pickup_period>60</suggested_pickup_period>
        <location>Chicago / Waukegan, Waukegan Regional Airport, IL</location>
        <station_id>KUGN</station_id>
        <latitude>42.420</latitude>
        <longitude>-87.870</longitude>
        <observation_time>Last Updated on Jul 28, 2:52 pm CDT</observation_time>
        <observation_time_rfc822>Sat, 28 Jul 2007 14:52:00 -0500 CDT</observatio
n_time_rfc822>
        <weather>Fair</weather>
        <temperature_string>76 F (24 C)</temperature_string>
        <temp_f>76</temp_f>
        <temp_c>24</temp_c>
        <relative_humidity>72</relative_humidity>
        <wind_string>From the Northeast at 12 MPH</wind_string>
        <wind_dir>Northeast</wind_dir>
        <wind_degrees>30</wind_degrees>
        <wind_mph>11.5</wind_mph>
        <wind_gust_mph>NA</wind_gust_mph>
        <pressure_string>29.98&quot; (1014.1 mb)</pressure_string>
        <pressure_mb>1014.1</pressure_mb>
        <pressure_in>29.98</pressure_in>
        <dewpoint_string>66 F (19 C)</dewpoint_string>
        <dewpoint_f>66</dewpoint_f>
        <dewpoint_c>19</dewpoint_c>
        <heat_index_string>78 F (26 C)</heat_index_string>
        <heat_index_f>78</heat_index_f>
        <heat_index_c>26</heat_index_c>
        <windchill_string>NA</windchill_string>
        <windchill_f>NA</windchill_f>
        <windchill_c>NA</windchill_c>
        <visibility_mi>10.00</visibility_mi>
        <icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base
>
        <icon_url_name>skc.jpg</icon_url_name>
        <two_day_history_url>http://www.weather.gov/data/obhistory/KUGN.html</tw
o_day_history_url>
        <ob_url>http://www.nws.noaa.gov/data/METAR/KUGN.1.txt</ob_url>
        <disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
        <copyright_url>http://weather.gov/disclaimer.html</copyright_url>
        <privacy_policy_url>http://weather.gov/notice.html</privacy_policy_url>
</current_observation>

INET>
```

Figure 15. Using wget to Read Un-Filtered RSS



```
!INFO! Connecting to Server - connected

<?xml version="1.0" encoding="ISO-8859-1"?>
<current_observation version="1.0"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://www.weather.gov/data/current_obs/
current_observation.xsd">
        <credit>NOAA's National Weather Service</credit>
        <credit_URL>http://weather.gov/</credit_URL>
        <image>
                <url>http://weather.gov/images/xml_logo.gif</url>
                <title>NOAA's National Weather Service</title>
                <link>http://weather.gov</link>
        </image>
        <suggested_pickup>15 minutes after the hour</suggested_pickup>
        <suggested_pickup_period>60</suggested_pickup_period>
        <location>Chicago / Waukegan, Waukegan Regional Airport, IL</location>
        <station_id>KUGN</station_id>
        <latitude>42.420</latitude>
        <longitude>-87.870</longitude>
        <observation_time>Last Updated on Jul 28, 2:52 pm CDT</observation_time>
        <observation_time_rfc822>Sat, 28 Jul 2007 14:52:00 -0500 CDT</observatio
n_time_rfc822>
        <weather>Fair</weather>
        <temperature_string>76 F (24 C)</temperature_string>
        <temp_f>76</temp_f>
        <temp_c>24</temp_c>
        <relative_humidity>72</relative_humidity>
        <wind_string>From the Northeast at 12 MPH</wind_string>
        <wind_dir>Northeast</wind_dir>
        <wind_degrees>30</wind_degrees>
        <wind_mph>11.5</wind_mph>
        <wind_gust_mph>NA</wind_gust_mph>
        <pressure_string>29.98&quot; (1014.1 mb)</pressure_string>
        <pressure_mb>1014.1</pressure_mb>
        <pressure_in>29.98</pressure_in>
        <dewpoint_string>66 F (19 C)</dewpoint_string>
        <dewpoint_f>66</dewpoint_f>
        <dewpoint_c>19</dewpoint_c>
        <heat_index_string>78 F (26 C)</heat_index_string>
        <heat_index_f>78</heat_index_f>
        <heat_index_c>26</heat_index_c>
        <windchill_string>NA</windchill_string>
        <windchill_f>NA</windchill_f>
        <windchill_c>NA</windchill_c>
        <visibility_mi>10.00</visibility_mi>
        <icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base
>
        <icon_url_name>skc.jpg</icon_url_name>
        <two_day_history_url>http://www.weather.gov/data/obhistory/KUGN.html</tw
o_day_history_url>
        <ob_url>http://www.nws.noaa.gov/data/METAR/KUGN.1.txt</ob_url>
        <disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
        <copyright_url>http://weather.gov/disclaimer.html</copyright_url>
        <privacy_policy_url>http://weather.gov/notice.html</privacy_policy_url>
</current_observation>

INET> wget <observation_time> http://www.weather.gov/data/current_obs/KUGN.xml



!INFO! Connecting to Server - connected

Last Updated on Jul 28, 2:52 pm CDT

Sat, 28 Jul 2007 14:52:00 -0500 CDT
INET> wget <temperature_string> http://www.weather.gov/data/current_obs/KUGN.xml



!INFO! Connecting to Server - connected

76 F (24 C)
INET> wget <wind_dir><wind_mph> http://www.weather.gov/data/current_obs/KUGN.xml



!INFO! Connecting to Server - connected

Northeast
11.5
INET> █
```

Figure 16. Using wget and Filter to Parse XML Tags



```
INET>
INET>
INET>
INET>
INET>
INET>
INET> wget http://www.weather.gov/data/current_obs/KUGN.rss



!INFO! Connecting to Server - connected

<?xml version="1.0" encoding="ISO-8859-1"?>

<rss version='2.0' xmlns:dc='http://purl.org/dc/elements/1.1/'>
     <channel>
<title>Weather at Chicago / Waukegan, Waukegan Regional Airport, IL - via NOAA's
 National Weather Service</title>
<link>http://www.weather.gov/data/current_obs/</link>
     <lastBuildDate>Sat, 28 Jul 2007 20:32:13 UT</lastBuildDate>
     <ttl>60</ttl>
     <description>Weather conditions from NOAA's National Weather Service. </des
cription>
     <language>en-us</language>
     <managingEditor>robert.bunge@noaa.gov</managingEditor>
     <webMaster>w-nws.webmaster@noaa.gov</webMaster>

     <image>
     <url>http://www.weather.gov/images/xml_logo.gif</url>
     <title>NOAA - National Weather Service</title>
     <link>http://www.weather.gov/data/current_obs/</link>
     </image>

      <item>
<title>Fair and 76 degrees F at Chicago / Waukegan, Waukegan Regional Airport, I
L</title>
<link>http://weather.noaa.gov/weather/current/KUGN.html</link>
<description>
<![CDATA[<img src="http://weather.gov/weather/images/fcicons/skc.jpg" class="noa
aWeatherIcon"  width="55" height="58" alt="Fair" style="float:left;" /><br />]]>

Winds are Northeast at 12 MPH.  The pressure is 29.98&quot; (1014.1 mb) and the
humidity is 72%.

         The heat index is 78.
         Last Updated on Jul 28, 2:52 pm CDT.
      </description>
      <guid isPermaLink="false">Sat, 28 Jul 2007 14:52:00 -0500 CDT</guid>
      </item>
      </channel>
      </rss>
INET>
INET>
INET>
INET> wget <title><description> http://www.weather.gov/data/current_obs/KUGN.rss



!INFO! Connecting to Server - connected


Weather at Chicago / Waukegan, Waukegan Regional Airport, IL - via NOAA's Nation
al Weather Service
Weather conditions from NOAA's National Weather Service.

NOAA - National Weather Service

Fair and 76 degrees F at Chicago / Waukegan, Waukegan Regional Airport, IL
 Winds are Northeast at 12 MPH. The pressure is 29.98 (1014.1 mb) and the humidi
ty is 72%. The heat index is 78.Last Updated on Jul 28, 2:52 pm CDT.
INET>
INET>
INET>
INET>
INET>
```

Figure 17. Using wget and Filter on a RSS Feed (Top Unfiltered, Bottom Filtered)

10 RSS/XML Feed Reader Embedded Appliance

The RSS/XML feed reader is an embedded appliance that allows you to display and hear real-time content from the World Wide Web. The embedded appliance provides instant real-time information without booting a PC. This appliance connects to the web, gets the desired feed, and parses the text information or character data from the feed. That data is displayed on the LCD, the serial port, and spoken through the text to speech processor.

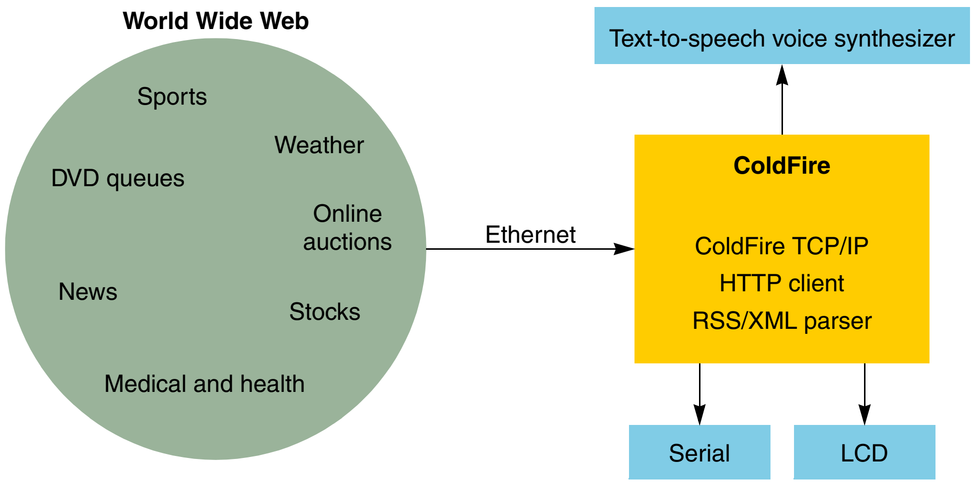


Figure 18. RSS/XML Feed Reader Block Diagram

11 RSS/XML Feed Reader Hardware

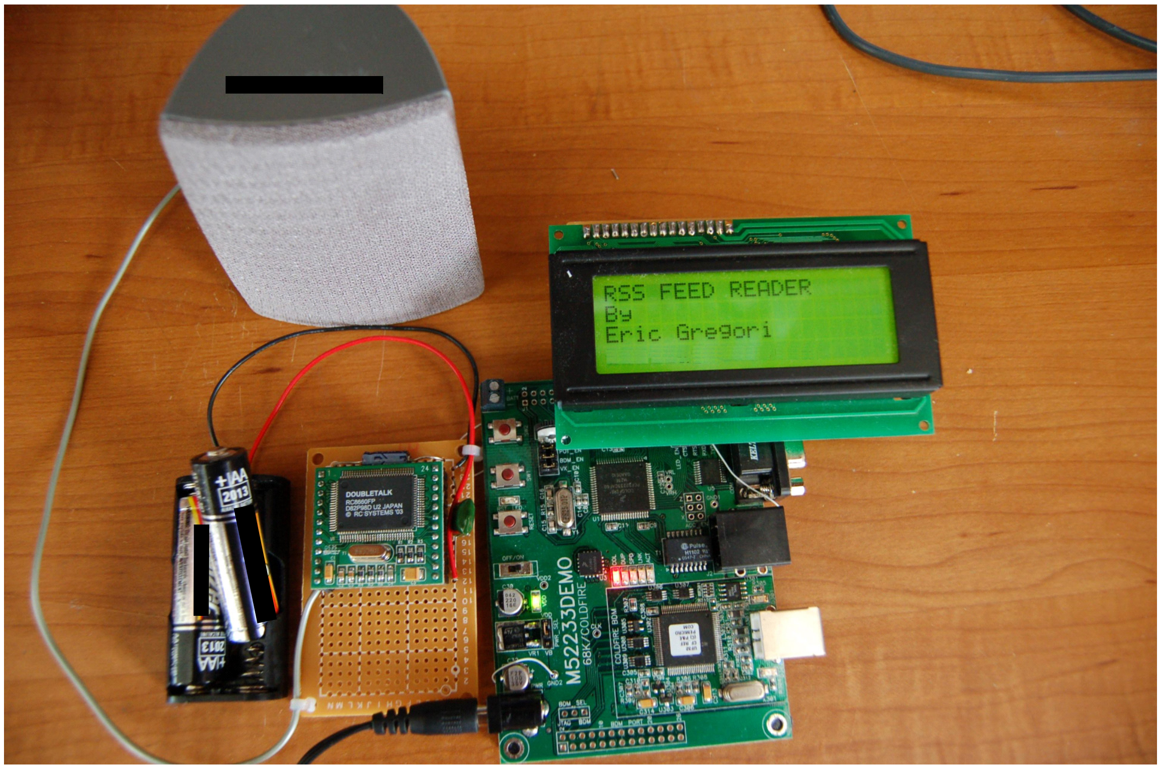


Figure 19. RSS/XML Feed Reader Hardware

11.1 M52233DEMO Board from Freescale Semiconductor

The M52233DEMO board is a reference board that evaluates the ColdFire MCF52233 processor. The demo board includes a serial port, USB BDM debug port, and an Ethernet port. The board along with the free (up to 128 KB of flash) CodeWarrior tools are all you need to begin working on your Ethernet projects.

This document runs Freescale's free public-source TCP/IP stack (available at http://www.freescale.com) on the demo board. The ColdFire TCP/IP stack is documented thoroughly in *ColdFire TCP/UDP/IP Stack and RTOS*, AN3470.

The demo board includes a 40-pin header connector to access most of the signals from the ColdFire microcontroller. The board also includes a three-axis accelerometer connected to three of the microcontroller's analog ports, a potentiometer, and two user buttons.

11.2 Interfacing to a Parallel LCD

The parallel LCD is a 4 × 20 character display that uses the standard Hitachi instruction set. The LCD is used in four-bit mode, requiring only six connections to the microcontroller: four-bit data bus, a clock signal (E), and a register select line (RS). The read/write line (RW) is tied to ground for write-only (there is no reason to read from the display). The display chosen for this document only requires one E clock. Some larger displays require two.

The LCD used is a 5 V display. Because the microcontroller is 3.3 V, some level shifting is required. This is accomplished using a simple transistor switch circuit, as shown in Figure 20. Unfortunately, this circuit is also an inverter. The software must use inverted logic when communicating with the LCD.

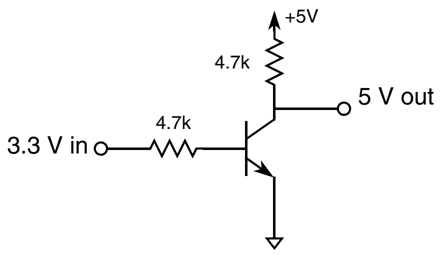


Figure 20. Simple 3.3 V to 5 V Signal Converter

11.2.1 ColdFire to 5 Volt LCD Interface

Table 5. ColdFire Demo Board to LCD Connection Table

LCD Pin	LCD Signal	ColdFire Pin	40 Pin J1 on Demo board	Power
1	Vss	GND	3	GND
2	Vcc	No connect	No connect	5 Volts
3	Vee	No connect	No connect	Variable
4	RS	GPT2	30	—
5	R/W	GND	3	—
6	E	GPT3	32	—
7	DB0	No connect	No connect	—
8	DB1	No connect	No connect	—
9	DB2	No connect	No connect	—

Table 5. ColdFire Demo Board to LCD Connection Table (continued)

LCD Pin	LCD Signal	ColdFire Pin	40 Pin J1 on Demo board	Power
10	DB3	No connect	No connect	—
11	DB4	Tin 0	34	—
12	DB5	Tin 1	36	—
13	DB6	Tin 2	38	—
14	DB7	Tin 3	40	—
15	LED+	No connect	No connect	4.2 Volts
16	LED-	GND	3	—

11.2.2 LCD Interface Board Schematic

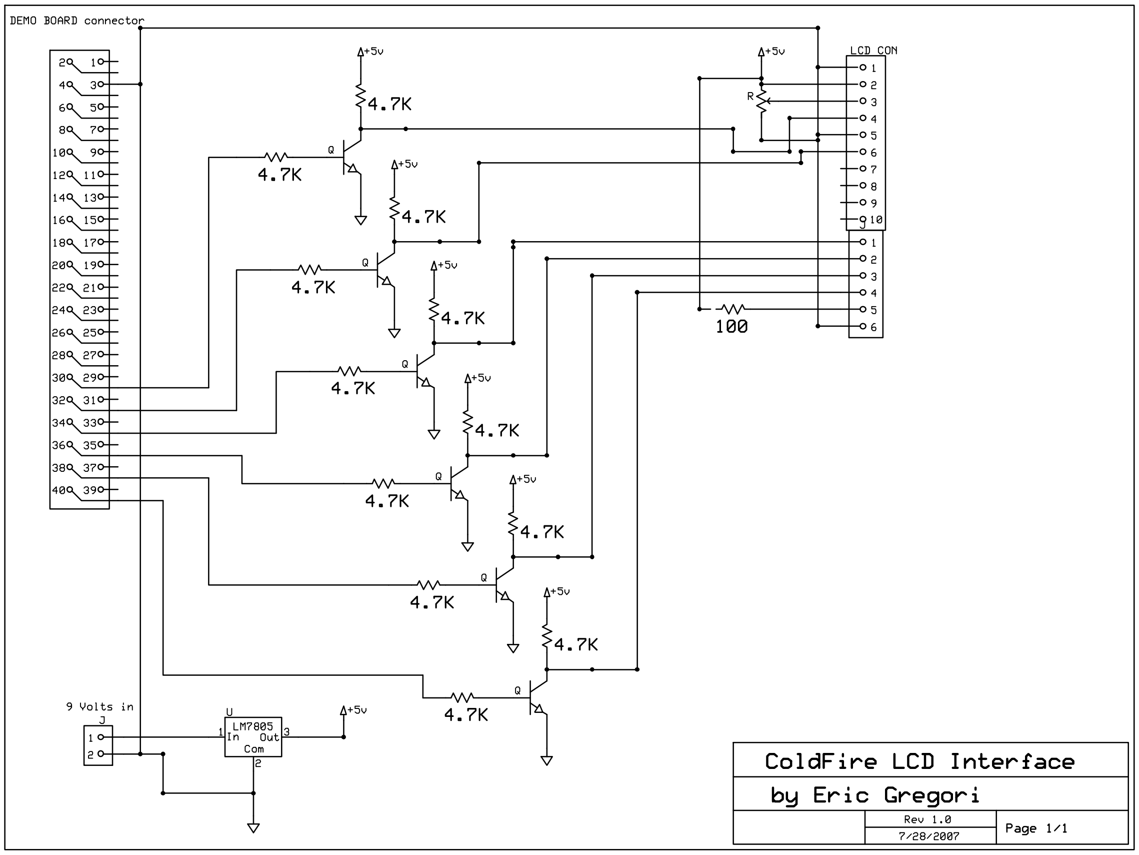


Figure 21. LCD Interface Schematic

11.2.3 LCD Interface Firmware

The heart of the LCD driver is the WriteToLCD function. This function sends character data or control data to the LCD over the GPIO bus. The data to write to the LCD is put on the bus, along with a signal (RS) indicating if the data is a character to be displayed or a control character. Then the E clock signal is pulsed to latch the data into the LCD controller. This must be done twice to load a full byte, upper nibble first, and then lower nibble.

The WriteToLCD function loads a single nibble each call. The function takes two arguments: the data nibble to send to the display (lower nibble) and whether the data is control or character data.

```
//*************************************************************************************
//* Function: void WriteToLCD(unsigned char data, unsigned char rs)
//*************************************************************************************
//* Author: Eric Gregori - Freescale FAE (Chicago)
//*
//*       Perform LCD initialization sequence
//*
//* Rev #    Date       Who        Comments
//* -----  -----------  ------     --------------------------------------------
//*  1.0    23-Jun-07   E.Gregori  Initial code release
//*************************************************************************************
*
void WriteToLCD(unsigned char data, unsigned char rs)
{
        volatile unsigned longdelay;

        // Write data to bus
        board_led_display(~data);

        for(delay=LCD_SHORT_DELAY; delay; delay--);

        // bring E clock high, set RS if requested
        if(rs)
                board_gpt_display(LCD_RS_HIGH_E_HIGH);
        else
                board_gpt_display(LCD_E_HIGH);

        // E clock high delay
        for(delay=LCD_SHORT_DELAY; delay; delay--);

        // bring E clock low
        if(rs)
                board_gpt_display(LCD_RS_HIGH);
        else
                board_gpt_display(LCD_E_LOW);

        // Data latches on falling edge of E clock,
        for(delay=LCD_SHORT_DELAY; delay; delay--);
}
```

All other LCD functions sit on top of this low-level driver. The LCD firmware is a driver stack with the low-level hardware control at the bottom and the higher level function on top.

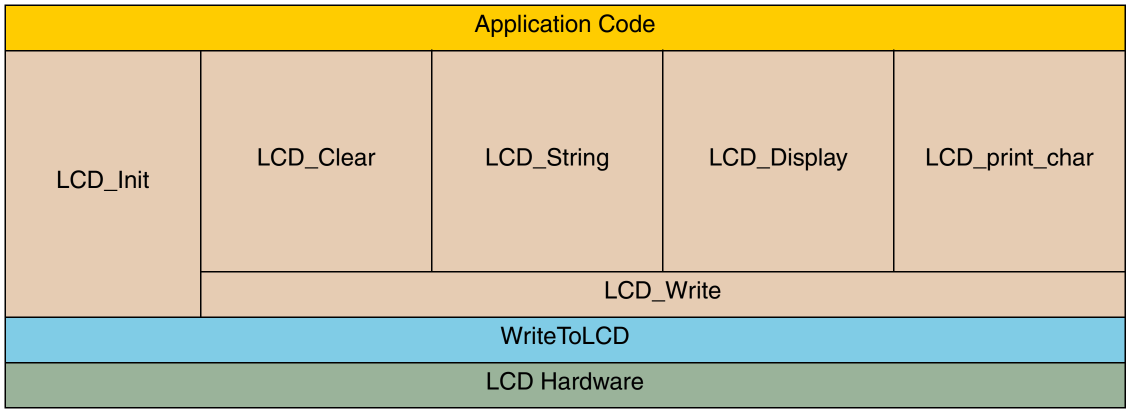


Figure 22. LCD Driver Stack

11.2.4 LCD Driver API

11.2.4.1 Function LCD_Init

```
Void LCD_Init (Void)
```

Initializes the hardware to communicate with the LCD. Must be called before calling any other function in the driver.

11.2.4.2 Function LCD_Clear

```
Void LCD_Clear (Void)
```

Clears the entire screen and set the cursor to line 4 (the bottom display line).

11.2.4.3 Function LCD_String

```
Void LCD_String (unsigned char *string,
                 unsigned char line)
```

Where	Equals
string	A pointer to a NULL terminated string
line	The line on the LCD to send the string to • Line 1 is at the top of the display • Line 4 is at the bottom of the display

Displays a NULL terminated string on the display line specified in parameter 2.

11.2.4.4 Function LCD_Display

```
Void LCD_Display(unsigned char *string,
                 unsigned char line,
                 unsigned char length)
```

Where	Equals
string	A pointer to a string of characters
line	The line on the LCD to send the string to • Line 1 is at the top of the display • Line 4 is at the bottom of the display
length	The number of bytes to display

Display a string of characters, on the specified line. The string of characters does not have to be null-terminated.

11.2.4.5 Function lcd_print_char

```
Void lcd_print_char(unsigned char data)
```

Where	Equals
data	character to add to the display

Displays data from left to right on the bottom line of the display. Lines scroll up when the bottom line is full, giving the appearance of a scrolling display. LCD_Clear must be called before sending the first byte of data. This function is useful for displaying a large stream of data.

11.3 Interfacing to a RCSystems V-Stamp Voice Synthesizer

The RCSystems V-Stamp Voice Synthesizer is an easy-to-use text-to-speech processor. The V-Stamp is a fully self-contained module, requiring only power, a speaker, a resistor, two capacitors, and a serial connection to an embedded system. The V-Stamp communicates with the embedded system using a UART. The module automatically sets its baudrate to that of the embedded system. From a hardware and firmware point of view, there is little work required to add the V-Stamp module to the RSS Feed Reader.

11.3.1 Voice Synthesizer Hardware

The V-Stamp requires two capacitors, a resistor, and a speaker of course. The sample application in the V-Stamp user manual was used as the voice synthesizer hardware circuit. The V-Stamp is powered by a external 3.3 volt power supply.

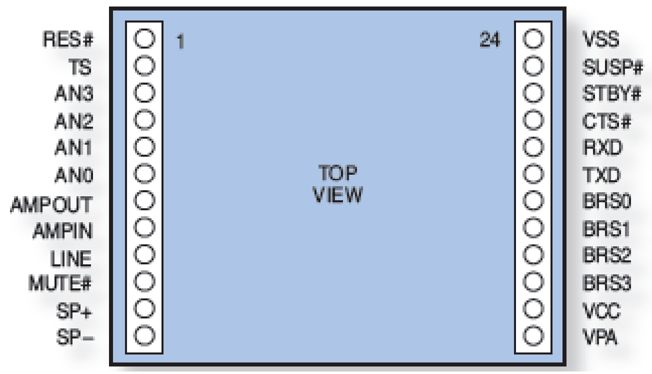


Figure 23. V-Stamp Module Pinout

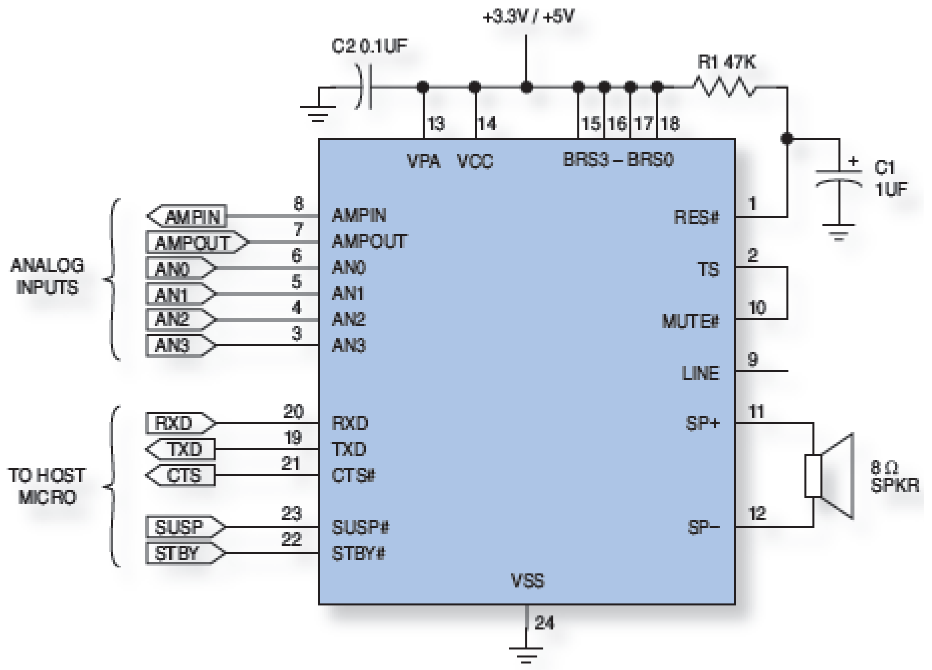


Figure 24. V-Stamp Typical Application Circuit

The UTXD0 line from the ColdFire is connected to the RXD line (pin 20) on the V-Stamp. The rest is up to software.

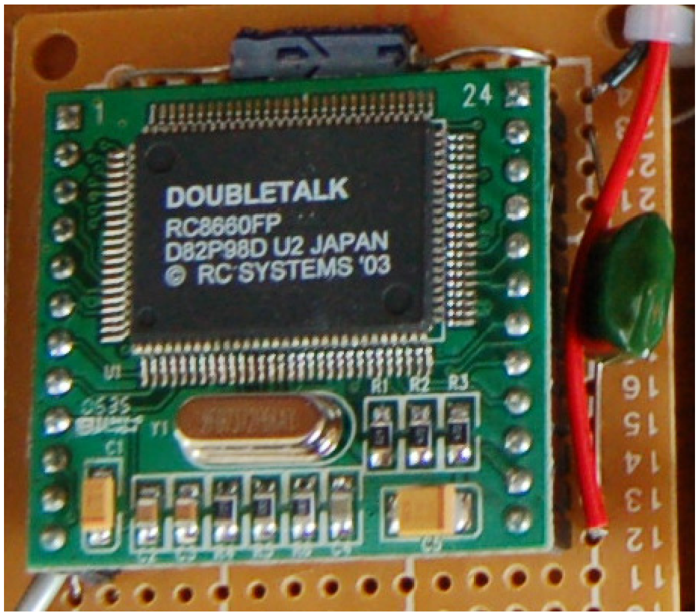


Figure 25. V-Stamp Hardware

11.3.2 Controlling the Text-to-Speech Processor

Out of reset, the V-Stamp starts converting text to speech. For this project, the default speech configuration is modified. There are many commands available to configure the V-Stamp, and all are covered in detail in the *RC8660 User Manual*.

Configuration commands start with a CTRL-A (0x01) followed by an ASCII sequence. Immediate commands use a single CTRL character: CTRL-P (0x10), CTRL-R (0x12), and CTRL-S (0x13). These commands can be intermixed with the text to convert. The next section discusses the configuration parameters changed from default.

11.3.3 Changing the Voice

The RC8660 supports 11 standard voices. Unfortunately, there is no good way to describe the different voices other then hearing them. Precise Pete sounds very clear. Robo Robert sounds very much like HAL from *2001: A Space Odyssey*. The Vader voice sounds exactly as the name implies (without the breathing). The O command is used to change voices. The O is preceded with a number from 0 to 10 to select the voice. All configuration commands start with a CTRL-A (0x01).

The command sequence is: CTRL-A (0x01) nO, where n = 0–10.

Table 6. Voice Options

Command	Voice Name
CTRL-A (0x01) 0O	Perfect Paul (default)
CTRL-A (0x01) 1O	Vader
CTRL-A (0x01) 2O	Big Bob
CTRL-A (0x01) 3O	Precise Pete
CTRL-A (0x01) 4O	Ricochet Randy
CTRL-A (0x01) 5O	Biff
CTRL-A (0x01) 6O	Skip
CTRL-A (0x01) 7O	Robo Robert
CTRL-A (0x01) 8O	Goliath
CTRL-A (0x01) 9O	Alvin
CTRL-A (0x01) 10O	Gretchen

11.3.4 Changing the Speed

The speech rate can be adjusted in 14 steps. Zero is the slowest, and thirteen is the fastest. The default is five. This application note slows the speech down to two. The S command adjusts the speech rate. The S is preceded with a number from 0 – 13 to select the rate. All configuration commands start with a CTRL-A (0x01).

The command sequence is: CTRL-A (0x01) nS, where n = 0 – 13 (zero is the slowest, and thirteen is the fastest).

11.3.5 Changing the Volume

The speech volume can be controlled by software. It is useful to quiet down the speech processor while the ColdFire TCP/IP stacks dumps all of its diagnostic data coming out of reset. The volume can be adjusted from 0 – 9. The default volume is 5. The V command is used to adjust the volume. The V is preceded with a number from 0 – 9, with nine being the loudest. All configuration commands start with a CTRL-A (0x01).

The command sequence is: CTRL-A (0x01) nV, where n = 0–9 (0 is lowest, 9 is highest).

11.3.6 Flushing the Text Buffer

The RC8660 has a large UART/text buffer (8 KByte). This allows the text-to-speech processor to work without interfereing with the operation of the embedded system. Most of the time the embedded system does not have to be concerned with overflowing the RC8660's UART buffer. The CTRL-X command flushes the buffer. Use this command to flush out all the ColdFire TCP/IP stack reset diagnostic data text from the RC8660's 8-KByte buffer. CTRL-A is not used with this command. This command is a single byte with no parameters.

The command sequence is: CTRL-X (0x18)

11.3.7 Voice Synthesizer Configuration Firmware

```
void emg_rss_reader_init(void)
{
        LCD_Init;
        LCD_String((unsigned char *)RSS FEED READER , 1);
        LCD_String((unsigned char *)By , 2);
        LCD_String((unsigned char *)Eric Gregori , 3);
        LCD_String((unsigned char *)www.emgware.com , 4);
        printf(\x18); // Stop/Flush Talker
        tk_sleep(200);
        printf(\x01);// CTRL-A
        printf(3O);    // Select voice
        printf(\x01);// CTRL-A
        printf(2S);    // Slow down voice
        printf(\x01);// CTRL-A
        printf(5V);    // 5 volume
        tk_sleep(2*200);
        printf(RSS feed reader by Eric Gregori\n);
        printf(RSS feed reader by Eric Gregori\n);
        tk_sleep(10*200);
}
```

12 RSS/XML Feed Reader Firmware

The RSS/XML feed reader firmware builds on top of all the other firmware described so far in this application note, as shown in Figure 26. The HTTP client is used, along with the DNS client to get the RSS or XML data. The data is passed through the callback to EMG_rss_text_filter that parses out the tags defined in the tag_filter array.

For the RSS reader, the tags are hardcoded to <title> and <description>. The big difference between the feed reader and the wget implementation described above is what happens to the data after it is filtered. In the wget implementation, the data was sent directly to the serial port. The RSS/XML reader sends the filtered character data to a character_buffer array.

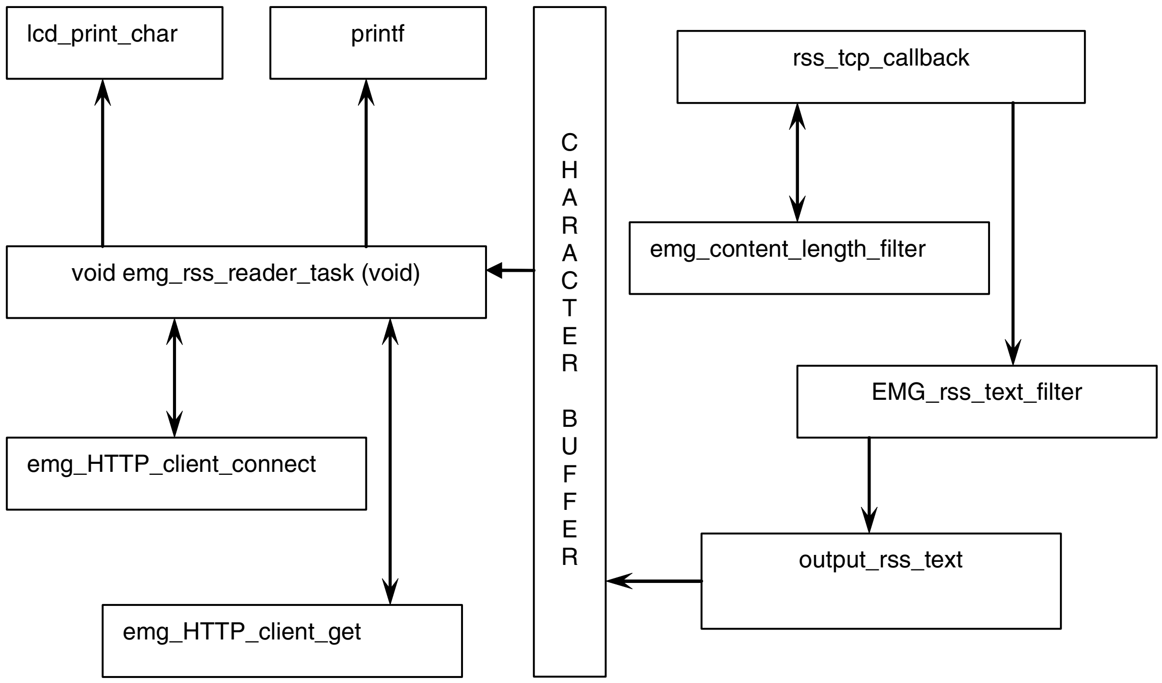


Figure 26. In the Module RSS_reader_byEricGregori.c

12.1 Character_buffer

All the filtered data is sent to the character buffer, including any additional newlines between tags. This buffering is necessary to keep the LCD display readable. If the data to the LCD is not slowed down, it scrolls too fast to read. The same applies to the speech synthesizer. The output_rss_text function call originates from the rss_tcp_callback function. The callback is called directly from the TCP/IP stack and must be treated as an interrupt. You do not want to delay or sleep in a callback function, or any function called from a callback function.

For this reason, the data from the filter is stored in the character buffer. Then, the emg_rss_reader_task can slowly feed the data from the buffer to the LCD and speech synthesizer without interfering with the callback timeing (and ultimately the TCP/IP stacks).

Putting filtered data into the character buffer:

```
//*****************************************************************************
// output_rss_text - Written By Eric Gregori
//                       eric.gregori@freescale.com
//
//       Called by emg_rss_text_filter
//
//*****************************************************************************
void output_rss_text(unsigned char data)
{
         if(character_buffer_index < RSS_CHARACTER_BUFF_SIZE)
         {
                 character_buffer[ character_buffer_index++ ) = data;
         }
}
```

Displaying and printing data from the character buffer:

```
emg_HTTP_client_close(shandle);

if(character_buffer_index)
{
        for(e=0; e<character_buffer_index; e++)
        {
                // Display Data from buffer
                lcd_print_char(character_buffer[e]);
                printf(%c, character_buffer[e]);
                if(character_buffer[e] == 0x0d)
                        tk_sleep(50);
        }
}
```

12.2 Using the RSS/XML Feed Reader

Using the RSS/XML feed reader firmware is easy:

1. Set the url variable to the URL or the desired RSS or XML server.

```
static const unsigned char url[] = “http://www.weather.gov/data/current_obs/KUGN.rss”;
```

2. Set the tag_filter() to the type of data you want to display.
   — For RSS feeds, <title> and <description> would be a first choice.
   — For XML feeds, this could be any tag name depending on the desired information.

```
const unsigned char *tag_filter() =
{
    {(const unsigned char *)”title”},
    {(const unsigned char *)”description”},
    {(const unsigned char *)””}
};
```

3. Set the character buffer size big enough to collect your filtered data.

```
#define RSS_CHARACTER_BUFF_SIZE 2048
```

4. Compile the project and flash it to the board.

After the TCP/IP stack comes up, the following occurs:

1. The DHCP client automatically aquires a IP address and DNS IP addresses.
2. A title screen is displayed and spoken.
3. The firmware connects to the server specified in the URL.
4. The status of the connection is displayed and spoken.
5. The file is downloaded from the server using the HTTP client.
6. The file is displayed and spoken after the connection closes.
7. The RSS/XML feed reader sleeps waiting for SW1 or SW2 to be pushed.
8. After SW1 or SW2 is pushed, a connection to the server specified in the URL is initiated and the process repeats.

12.3 XML Streams

For XML streams, the EMG_rss_text_filter return value determines which tag the data is from. This allows another const array containing more descriptive names to describe the data from the tag. Use the EMG_rss_text_filter return value minus one to index into a descriptive name array and send the indexed string into the character buffer before leaving the RSS callback function. The XML filter then places the data from the tag in the character filter directly after the descriptive name.

THIS PAGE IS INTENTIONALLY BLANK

Document Number: AN3518
Rev. 0
09/2007

freescale™
semiconductor