Cika Electrónica

Nota de Aplicación: CAN-006
Título: **Pizarra remota via HTTP con Rabbit 2000 y LCD gráfico (SED1335)**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	16/7/03	

Desarrollamos una simple aplicación de un módulo LCD gráfico inteligente y un módulo Rabbit conectado a una red Ethernet. Se trata de una "pizarra remota" que puede escribirse via HTTP. El módulo Rabbit funciona como un servidor web y con cualquier navegador podemos escribir un simple mensaje en la pizarra. Más allá de su posible utilidad en la vida real, esta aplicación nos permite ejemplificar el uso de las capacidades de TCP/IP de Dynamic C y aportar algo más sobre el display.

Como display, utilizaremos un módulo Powertip PG320240, de 320x240 pixels, basado en chips controladores compatibles con el SED1335, de S-MOS, y su clon de EPSON. No se darán demasiados detalles acerca del display y su software de control, dado que este tema se ha desarrollado en la CAN-005. El lector puede remitirse a dicha nota de aplicación para mayor información.

El código de la pizarra en sí, resulta muy similar al desarrollado en la CAN-004 para el display de 128x64; no obstante, lo desarrollamos en su totalidad para evitar confundir al lector.

Hardware

Las conexiones del display son las que figuran en la tabla a la derecha:

Rabbit	LCD
PA.0	D0
PA.1	D1
PA.2	D2
PA.3	D3
PA.4	D4
PA.5	D5
PA.6	D6
PA.7	D7
PE.4	RD
PE.3	WR
PE.0	A0
PE.1	CS

El port A, hace las veces de bus de datos, mientras que los ports libres del port E generarán, por software, las señales de control. El port Ethernet ya viene funcionando en el módulo, no debemos hacer nada más que conectarlo a una red Ethernet. Se recomienda la lectura de la CAN-005 para detalles sobre el conexionado y manejo de los pines de RST y CS.

Descripción del funcionamiento

Nuestra pizarra será un display gráfico de 320x240, que mostrará texto en 30 filas de 40 caracteres. La fila superior mostrará la fecha y hora, y los mensajes ingresarán por la fila inferior, desplazando el texto anterior hacia arriba. El controlador del display incluye caracteres de 5x7, definiremos nuestro propio set de caracteres de 8x8 para una mejor lectura.

Para ingresar un mensaje, el usuario se conecta con su navegador preferido a nuestra dirección IP, y verá una planilla en la que ingresará el texto a mostrar. El usuario puede ver además la fecha y hora y la cantidad de mensajes enviados. Esto se realiza mediante "server side includes" (SHTML).

La planilla es un HTML FORM, en modo POST. Al remitirla (submit), el usuario hace que el módulo Rabbit ejecute una función (CGI), cuyos parámetros son provistos por el navegador y su resultado es la actualización del display; al mismo tiempo que se le mostrará en el navegador una segunda pantalla, confirmando la acción.

La información de fecha y hora se toma directamente del timer provisto por Dynamic C, se asume que el usuario ya hizo la "puesta en hora" del sistema mediante alguno de los ejemplos provistos por Dynamic C. No es necesario reajustar fecha y hora a menos que se apague el módulo y no se disponga de battery backup.

Software

Desarrollaremos a continuación el software de la pizarra remota. Tanto el código en sí como el uso de HTML son simples, y están lejos de ser óptimos; su misión fundamental es demostrar la facilidad de uso de las bibliotecas de funciones que provee Dynamic C.

Comenzamos por la inicialización de constantes de TCP/IP. Si bien el método utilizado en la presente nota de aplicación es algo antiguo, y existe una nueva forma para inicializar estos parámetros, preferimos utilizarlo debido a su extrema simpleza, que lo hace particularmente atractivo para los no-programadores.

En lo que al procesamiento HTML se refiere, si bien se recomienda poseer cierta familiaridad con la terminología y funcionamiento de un web server, la misma no es estrictamente necesaria para comprender el desarrollo de la mayoría de las funciones.

Como en cualquier host conectado a una red TCP/IP, comenzamos definiendo la dirección IP , la máscara, y la dirección del router; este último sólo es necesario si vamos a acceder al módulo desde una red ruteada, además de nuestra red local. Si bien Dynamic C soporta obtención de la dirección IP mediante DHCP, dado que se trata de un web server, su dirección debería ser fija, por lo que preferimos dejar el DHCP para otra oportunidad. La dirección IP a utilizar deberíamos coordinarla con el administrador de la red, a menos, claro está que tengamos nuestro propio segmento.

```
/* TCP/IP stuff */

#define MY_IP_ADDRESS   "192.168.1.52"
#define MY_NETMASK      "255.255.255.0"

#define MY_GATEWAY            "192.168.1.1"         // only needed if out of local LAN
```

A continuación incluímos las bibliotecas de funciones que soportan TCP/IP y HTTP respectivamente. Dada la gran cantidad de código, probablemente no quepa en el segmento base, por lo que habilitamos el uso de XMEM. De esta forma, el compilador ubicará las funciones en memoria física para ser utilizadas dentro del segmento XMEM, e insertará el manejo de la MMU para accederlas, todo sin nuestra intervención. Esto se realiza de forma muy simple mediante la directiva *#memmap*.

```
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
```

Para almacenar las páginas web, utilizaremos la directiva *#ximport*, que nos permite leer un archivo al momento de compilar y guardarlo en memoria (XMEM), pudiendo referenciarse mediante un puntero:

```
/* use flash import facility */

#ximport "bb.shtml"      index_html
#ximport "ok.html"      ok_html
#ximport "rabbit1.gif"   rabbit1_gif
```

Así, index_html es un puntero a la posición de memoria física donde está guardado el archivo *bb.shtml*, tal cual figuraba en el directorio de nuestra máquina al momento de compilar el proyecto. El path especificado es relativo al archivo fuente de donde se lo llama.

A continuación, debemos definir los tipos MIME y su correspondiente handler. Esto se debe a que no tenemos un sistema operativo que nos resuelva estas tareas, y debemos decirle al servidor qué es cada cosa, es decir, algo así como la función desempeñada por el archivo mime.types en Linux.

Cuando el URL no especifica archivo, como por ejemplo http://alguna_dirección/ , el servidor generalmente entrega el archivo que se le especifica en su configuración, generalmente index.html. En este caso, debemos primero indicar el tipo MIME para ese caso en particular, por lo que la primera entrada en la estructura corresponde al acceso al URL sin especificar un archivo:

```
/* the default for / must be first */
const static HttpType http_types[] =
{
   { ".shtml", "text/html", shtml_handler}, // ssi
   { ".html", "text/html", NULL},           // html
   { ".cgi", "", NULL},                     // cgi
   { ".gif", "image/gif", NULL}
};
```

Así, definimos que un acceso a http://MY_IP_ADDRESS/ es un acceso a un archivo de tipo SHTML, que los archivos terminados en *.shtml* y *.html* serán reportados por el servidor como de tipo (MIME type) *text/html*, mientras que los terminados en *.gif* se reportarán como *image/gif*. Asimismo, definimos que los archivos de tipo SHTML serán procesados por un handler llamado *shtml_handler*, que es el engine para los server side includes que provee Dynamic C, y el resto de los otros tipos definidos utilizará el handler por defecto. La entrada correspondiente a archivos de tipo CGI, *.cgi*, simplemente define la extensión, definiremos cómo se procesa la función más adelante.

Definimos algunas variables globales, que accederemos desde varias funciones:

```
int msg;                                    // count number of messages
char date[11],time[9];                      // store current date&time
```

Definimos la función que usaremos para ejecutar nuestro CGI, ya que la vamos a necesitar a continuación:

```
int submit(HttpState*);
```

Ahora, debemos decirle al servidor HTTP (el cual Dynamic C provee listo para nuestro uso) de qué archivos dispone para trabajar, es decir, asociamos los URLs con los punteros a la información que importamos antes con *#ximport*. En este caso utilizaremos la forma más fácil, que consiste en aprovechar una estructura definida en HTTP.LIB, la biblioteca de funciones del web server, llamada *http_flashspec*:

```
const static HttpSpec http_flashspec[] =
{
   { HTTPSPEC_FILE,  "/",             index_html,    NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/index.shtml",  index_html,    NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/ok.html",      ok_html,       NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/rabbit1.gif",  rabbit1_gif,   NULL, 0, NULL, NULL},

   { HTTPSPEC_VARIABLE, "msg", 0,     &msg, INT16,   "%d", NULL},
   { HTTPSPEC_VARIABLE, "date", 0,    date, PTR16,   "%s", NULL},
   { HTTPSPEC_VARIABLE, "time", 0,    time, PTR16,   "%s", NULL},

   { HTTPSPEC_FUNCTION, "/submit.cgi", 0,            submit, 0, NULL, NULL}
};
```

Las primeras cuatro entradas asocian los siguientes URLs con los punteros y por ende archivos que figuran a continuación (ver *#ximport* más arriba):

http://MY_IP_ADDRESS/ ó http://MY_IP_ADDRESS/index.shtml -> index_html, puntero a bb.shtml
http://MY_IP_ADDRESS/ok.html -> ok_html, puntero a ok.shtml
http://MY_IP_ADDRESS/rabbit1.gif -> rabbit1_gif, puntero a rabbit1.gif

Cabe recordar que el contenido de dichos archivos fue copiado y asociado al puntero al utilizar *#ximport*. A su vez, le dijimos al servidor cómo debía manejar cada tipo de archivo al definir *http_types* (ver más arriba).

Las tres líneas siguientes definen tres variables a ser procesadas por el servidor al recibir una petición de una página SHTML (HTTP GET). Definimos el nombre como se la refiere en el SHTML, la variable en el programa (global), el tipo, y la forma de tratarla para mostrar su valor:

- la variable *msg* es un entero (int), alojado en *&msg*, y se muestra en decimal.
- las variables *date* y *time* son strings, comenzando en las posiciones *date* y *time* respectivamente.

La última línea asocia el URL http://MY_IP_ADDRESS/submit.cgi con la función *submit*, a ser ejecutada cuando el usuario presione el botón asociado a la planilla (TYPE=SUBMIT, ACTION=/submit.cgi), es decir, una petición de ese URL produce la ejecución de la función *submit*, que desarrollaremos más adelante. La

función recibe un puntero a la estructura donde el servidor HTTP contiene toda la información necesaria como para que la función pueda procesar la información.

Por comodidad, al ejecutar el CGI preferimos realizar un HTTP REDIRECT en vez de manejar la entrega de la respuesta dentro de la función, lo cual implicaría escribir en el handler del port TCP. Al usar el redirect, simplemente le decimos al navegador que vaya a buscar otra página y allí se le presentará la confirmación de la operación. Por generalidad, definimos la acción de la siguiente manera:

```
#define REDIRECTHOST          MY_IP_ADDRESS
#define REDIRECTTO      "http://" REDIRECTHOST "/ok.html"
```

Hasta aquí hemos desarrollado toda la inicialización de nuestro servidor HTTP. Comenzamos ahora el procesamiento (parsing) de la información de la planilla (form) HTML. Algunas de las variables o funciones referidas aquí serán desarrolladas más adelante, por lo que deberíamos definirlas. En el archivo que tiene el código fuente, el orden de las funciones es distinto para evitar este problema. Preferimos aquí desarrollar el ejemplo en el orden que favorezca la comprensión.

Generalmente, el extraer información de un POST suele ser una tarea tediosa; cuando el usuario presiona el botón de enviar información (submit), el navegador se conecta al servidor y envía la información de la planilla codificada en un string. Nuestra tarea es analizar el string, buscando delimitadores conocidos, y extraer la información que nos interesa, es decir, el texto a mostrar en pantalla. Afortunadamente, uno de los ejemplos que provee Dynamic C muestra exactamente cómo realizar esta tarea:

Definimos un espacio para guardar los resultados (1), esperamos una cantidad de variables (1), extraeremos su contenido y lo guardaremos en el campo correspondiente de la estructura definida.

```
#define MAX_FORMSIZE   64
typedef struct {
       char *name;
       char value[MAX_FORMSIZE];
} FORMType;
FORMType FORMSpec[1];
```

Esta función extrae una variable a la vez. Tanto HTTP_MAXBUFFER como el tipo de variable HttpState han sido definidos al incluír la biblioteca de funciones de HTTP (http.lib).

```
/*
 * Parse one token 'foo=bar', matching 'foo' to the name field in
 * the struct, and storing 'bar' into the value
 */
void parse_token(HttpState* state)
{
       auto int i, len;

       for(i=0; i<HTTP_MAXBUFFER; i++) {
              if(state->buffer[i] == '=')
                     state->buffer[i] = '\0';
       }
       state->p = state->buffer + strlen(state->buffer) + 1;

       for(i=0; i<(sizeof(FORMSpec)/sizeof(FORMType)); i++) {
              if(!strcmp(FORMSpec[i].name,state->buffer)) {
                len=(strlen(state->p)>MAX_FORMSIZE) ? MAX_FORMSIZE-1 : strlen(state->p);
                strncpy(FORMSpec[i].value,state->p,1+len);
                FORMSpec[i].value[MAX_FORMSIZE-1] = '\0';
              }
       }
}
```

Esta función procesa el string, extrayendo a través de la función anterior, una a una todas las variables. El resultado sigue estando url-encoded, es decir, los caracteres conflictivos fueron reemplazados por secuencias de escape. Nos ocuparemos de esto más adelante.

```
/*
 * parse the url-encoded POST data into the FORMSpec struct
 * (ie: parse 'foo=bar&baz=qux' into the struct
 */
int parse_post(HttpState* state)
{
	auto int retval;

	while(1) {
		retval = sock_fastread((sock_type *)&state->s, state->p, 1);
		if(0 == retval) {
			*state->p = '\0';
			parse_token(state);

			return 1;
		}

		/* should this only be '&'? (allow the eoln as valid text?) */
		if((*state->p == '&') || (*state->p == '\r') || (*state->p == '\n')) {
			/* found one token */
			*state->p = '\0';
			parse_token(state);

			state->p = state->buffer;
		} else {
			state->p++;
		}

		if((state->p - state->buffer) > HTTP_MAXBUFFER) {
			/* input too long */
			return 1;
		}
	}

	return 0; /* end of data - loop again to give it time to write more */
}
```

Ahora, la función CGI: extraemos la información que nos manda el navegador del buffer, la procesamos y extraemos el texto a mostrar en pantalla, incrementamos el número de mensajes recibidos y devolvemos la pantalla de confirmación. La función *http_urldecode* es la que nos permite extraer la información del texto url-encoded que extrajeron las funciones anteriores. Dicha función viene incluída en la biblioteca de funciones de HTTP. Por simpleza, no incluímos manejo de errores, dado el caracter de demostración.

```
int submit(HttpState* state)
{
char buffer[MAX_FORMSIZE];
char num[5];

	FORMSpec[0].value[0] = 0;                          // inicializa variable (nada)
	if(parse_post(state)) {                            // extrae valor de variable
		if(*(http_urldecode(buffer,FORMSpec[0].value,MAX_FORMSIZE))) {
                                    // decodifica datos, ignora cadenas vacías
			buffer[36]=0;                          // string <= ancho display
			LCD_scroll();                          // mueve texto anterior arriba
			if (msg==99)
				msg=0;
			sprintf(num,"%2d: ",msg);
			LCD_printat(29,0,num);                 // muestra número de mensaje
			LCD_printat(29,4,buffer);              // muestra mensaje
```

```
                        msg++;                                  // actualiza contador de msgs
                }
                cgi_redirectto(state,REDIRECTTO);               // Muestra OK
        } else {
                                                                // insertar manejo de errores
        }
        return(0);
}
```

Y eso es todo lo que necesitamos hacer.

Veamos ahora someramente la inicialización y funciones de control del display. Sólo nos detendremos en aquellas que difieran de lo desarrollado en la CAN-005. Particularmente, la única diferencia significativa es la modificación de la secuencia de inicialización para utilizar CGROM externa. Mantuvimos el resto igual que en la CAN-005 a fin de no confundir con demasiadas modificaciones.

```
/* LCD control signals */
#define LCD_RD  4
#define LCD_A0  0
#define LCD_WR  3
#define LCD_CS  1

/* Low level functions */

#asm
;function requires one parameter:
;@sp+2= data to write
;
LCD_Write::
        ld hl,(sp+2)                ; extrae del stack el valor a escribir (LSB)
        ld a,l
        ioi ld (PADR),a             ; lo escribe, PA0-7 se mantienen como salidas
        ld hl,PEDRShadow            ; apunta al control del port E (shadow)
        ld de,PEDR                  ; apunta al control del port E
        res LCD_WR,(HL)             ; Baja WR
        ioi ldd                     ; ahora (atomic)
        ld hl,PEDRShadow            ; apunta al control del port E (shadow)
        ld de,PEDR                  ; apunta al control del port E
        set LCD_WR,(HL)             ; Sube WR
        ioi ldd                     ; ahora
        ret

#endasm


void LCD_WriteCmd(unsigned char cmd)
{
        BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_A0 );  // Sube A0 (Cmd)
        LCD_Write(cmd);
        BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_A0 );  // Baja A0 (Data)
}

void LCD_WriteStrCmd(unsigned char *cmd,int len)
{
        LCD_WriteCmd(*cmd++);                        // manda comando
        while(len--) {                               // manda parámetros
                LCD_Write(*cmd++);
        }
}

unsigned char LCD_ReadData()
{
unsigned char data;
        WrPortI ( SPCR, &SPCRShadow, 0x80 );         // PA0-7 = Inputs
        BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_A0 );  // Sube A0 (Data)
```

```
        BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_RD );  // Baja RD
        data=RdPortI(PADR);                          // lee datos del bus
        BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_RD );  // Sube RD
        BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_A0 );  // Baja A0 (Sts)
        WrPortI ( SPCR, &SPCRShadow, 0x84 );         // PA0-7 = Outputs
        return(data);
}

void MsDelay ( int iDelay )
{
        unsigned long ul0;
        ul0 = MS_TIMER;                              // valor actual del timer
        while ( MS_TIMER < ul0 + (unsigned long) iDelay );
}


void LCD_init ()
{
const static unsigned char init_string1[]={
0x40,                   // INIT
'\B00110001',           // CGROM externa (RAM en F000), no top-screen corr, LCD, normal
0x87,0x7,               // 8x8 characters
39,59,                  // 320 pixels, 8 pixel characters => 40 characters per line
239,                    // 240 lines (30 txt lines)
40,0                    // virtual screen = display screen
};

const static unsigned char init_string2[]={
0x44,                   // SCROLL
0,                      // SAD1L: screen 1 empieza en (0x0000)
0,                      // SAD1H:
239,                    // SL1  : scrolling lines (239 dec.)
0xB0,                   // SAD2L: screen 2 empieza en (1200=0x4b0)
0x04,                   // SAD2H:
239,                    // SL2  : scrolling lines
0,                      // SAD3L:
0,                      // SAD3H:
0,                      // SAD4L:
0                       // SAD4H:
};

const static unsigned char init_string3[]={
0x5D,                   // CSRFORM
0x04,                   // CRX: cursor horiz (4 pixel)
0x86                    // CRY: cursor vert (6 pixel) CM : block cursor
};


const static unsigned char init_string4[]={
0x5A,                   // HDOT_SCR (0x5a), scroll rate.
0                       // 1 pixel scroll
};

const static unsigned char init_string5[]={
0x5B,                   // OVLAY
'\B00000001'            // XOR screens 1 y 2
};

const static unsigned char init_string6[]={
0x59,                   // DISP_ON
'\B00010110'            // cursor flash = 2 Hz, SAD1/2 no flashing, SAD3 off
};

        WrPortI ( PEDR,&PEDRShadow,'\B00011010' );          // CS,RD,WR = HIGH
        WrPortI ( PEDDR,&PEDDRShadow,'\B10011011' );        // PE0,1,4,3,7 = output
        WrPortI ( SPCR, &SPCRShadow, 0x84 );                // PA0-7 = Outputs
```

```
        MsDelay ( 1000 );                                       // espera LCD reset

        BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_CS );             // Baja CS

        LCD_WriteStrCmd ( init_string1,sizeof(init_string1) ); // inicializa el display
        LCD_WriteStrCmd ( init_string2,sizeof(init_string2) );
        LCD_WriteStrCmd ( init_string3,sizeof(init_string3) );
        LCD_WriteCmd ( 0x4C );        // CSRDIR_RIGHT, cursor incrementa a la derecha
        LCD_WriteStrCmd ( init_string4,sizeof(init_string4) );
        LCD_WriteStrCmd ( init_string5,sizeof(init_string5) );
        LCD_WriteStrCmd ( init_string6,sizeof(init_string6) );
}


/* High level functions */

void LCD_cursor ( int address )
{
        LCD_WriteCmd(0x46);                                     // CSRW.
        LCD_Write(address&0xFF);                                // LO byte
        LCD_Write((address>>8)&0xFF);                           // HI byte
}

void LCD_fill(unsigned char pattern, int len)
{
int i;
        LCD_WriteCmd(0x42);                                     // MWRITE
        while(len--)
                LCD_Write(pattern);                             // llena con pattern
}

void LCD_cleargfx ( void )
{
        LCD_cursor(1200);             // direcciona la pantalla gráfica
        LCD_fill(0,9600);             // borra pantalla (llena con 0's)
}

void LCD_cleartxt ( void )
{
        LCD_cursor(0);                        // direcciona pantalla de texto
        LCD_fill(' ',1200);                   // borra pantalla (llena con espacios)
}

void LCD_printat (unsigned int row, unsigned int col, char *ptr)
{
        LCD_cursor (40*row+col);              // posiciona cursor
        LCD_WriteCmd(0x42);                   // MWRITE
        while (*ptr)                          // escribe caracteres
                LCD_Write (*ptr++);
}
```

Agregamos una nueva función, *scroll*, que nos permite desplazar el contenido de la pantalla hacia arriba para que el nuevo mensaje “empuje” a los anteriores hacia arriba:

```
void LCD_scroll()
{
int i,line;
unsigned char data[40];

        for(line=2;line<30;line++) {                // no modifica primer renglón
                LCD_cursor(40*line);                // 1er caracter en renglón
                LCD_WriteCmd(0x43);                 // MREAD
                for(i=0;i<40;i++)
                        data[i]=LCD_ReadData();     // lee renglón y copia en buffer
                if(line==29){
```

```
                        LCD_cursor(40*line);          // borra último renglón
                        LCD_WriteCmd(0x42);           // MWRITE
                        for(i=0;i<40;i++)
                                LCD_Write(' ');
                }
                LCD_cursor(40*(line-1));              // renglón de arriba
                LCD_WriteCmd(0x42);                   // MWRITE
                        for(i=0;i<40;i++)
                                LCD_Write(data[i]);   // copia buffer en renglón de arriba
        }
}
```

El manejo de textos incorpora una diferencia respecto a la la CAN-005: aprovechamos la opción de ROM de caracteres externa del SED1335 y el hecho de que, en esa posición de memoria, exista RAM en el display; con lo cual, escribiendo un set de caracteres en esa área, logramos que el controlador muestre nuestro set de caracteres de 8x8 en vez del interno, de 5x7. Obsérvese como indicamos al compilador que sitúe los datos en XMEM. La copia de XMEM al display se hace mediante un buffer en área base y copiando bloques, como hiciéramos con las imágenes en la CAN-005.

```
void LCD_loadfont()
{
xmem const static unsigned char fontdata[] = {
<eliminada por cuestiones de espacio, 8 bytes por cada caracter, 256 caracteres>
};
int x,y;
unsigned char buffer[256];

        LCD_cursor(0xF000);           // direcciona "CGROM" (en realidad hay RAM allí)
        LCD_WriteCmd(0x42);           // MWRITE
        for(y=0;y<8;y++){
                xmem2root(&buffer,(long) fontdata+256*y,256); // lee de xmem
                for(x=0;x<256;x++)
                        LCD_Write(buffer[x]);         // manda al display
        }
}
```

Agregamos ahora una simple función para dar formato a la información de fecha y hora

```
/* RTC functions */

void get_datetime()
{
struct tm thetm;

        mktm(&thetm, SEC_TIMER);
        sprintf(date,"%02d/%02d/%04d",thetm.tm_mday,thetm.tm_mon,1900+thetm.tm_year);
        sprintf(time,"%02d:%02d:%02d",thetm.tm_hour, thetm.tm_min, thetm.tm_sec);
}
```

A continuación, el cuerpo del programa principal. Para mayor claridad, y de paso mostrar cómo se aplican las primitivas de soporte de multitarea cooperativo de Dynamic C, manejaremos la actualización de la fecha y hora y el procesamiento TCP/IP en tareas separadas

```
/* MAIN PROGRAM */

main()
{

        msg=1;                                        // 1er msg = 1
        FORMSpec[0].name = "texto";   // el msg está en la variable texto de la forma HTML
```

```
   sock_init();                                    // inicializa TCP/IP
   http_init();                                    // inicializa HTTP server

        LCD_init();                                // inicializa display
        LCD_cleartxt();                            // borra pantalla de texto
        LCD_cleargfx();                            // borra pantalla gráfica
        LCD_loadfont();                            // carga font 8x8

        while (1) {                                // loop infinito

               costate {                    // define multitarea cooperativo, tarea 1
                      http_handler();              // atiende TCP/IP
                      yield;                       // devuelve control a otras tareas
               }

               costate {                    // define multitarea cooperativo, tarea 2
                      waitfor(DelaySec(1));        // espera que pase 1 seg
                      get_datetime();              // actualiza strings date y time
                      LCD_printat(0,10,date);      // muestra en el display
                      LCD_printat(0,22,time);
               }
        }
}
```

La función *costate* es la que nos define cada tarea. La función *waitfor* espera que su parámetro evalúe como cierto (true), caso contrario devuelve el control a las otras tareas. Dado que el parámetro es *DelaySec(1)*, el resultado concreto es que a cada paso por esa instrucción, se devuelve el control a las demás tareas hasta tanto haya transcurrido un segundo, momento en el cual se ejecuta el bloque a continuación y se reevalúa el loop.

Hasta aquí todo lo relacionado con el código en sí. Veremos ahora la parte HTML y SHTML que será procesada por el servidor HTTP (provisto por Dynamic C), sin intervención de nuestra parte.

Veamos *bb.shtml*, este archivo es SHTML e informa al servidor qué variables debe incluir para que la página se muestre actualizada cada vez que se la solicita

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head><title>Billboard</title></head>
<body bgcolor="#FFFFFF" link="#009966" vlink="#FFCC00" alink="#006666" topmargin="0"
leftmargin="0" marginwidth="0" marginheight="0">
```

Muestra el logo de Rabbit

```
<center><img SRC="rabbit1.gif" ></center>
```

Define la planilla en sí

```
<form ACTION="submit.cgi" METHOD="POST">
<table>
<tr>
```

Define las variables de fecha y hora

```
<th><!--#echo var="date"-->
<th><!--#echo var="time"-->
```

Define la variable que muestra número de mensajes

```
<tr>
<td WIDTH="20%">Mensaje #<!--#echo var="msg"-->
```

El servidor detecta `<!--#echo var="msg"-->` y lo reemplaza por el contenido de la variable *`msg`*

```
<td WIDTH="80%"><input TYPE="TEXT" NAME="texto" SIZE=20>
</table>
<input TYPE="SUBMIT" VALUE="Mostrar"></form>
</body>
</html>
```

El archivo ok.html es aún más simple, dado que solamente muestra OK en la pantalla.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
<HEAD>
<TITLE>Billboard OK</TITLE>
</HEAD>
<BODY topmargin="0" leftmargin="0" marginwidth="0" marginheight="0"
       bgcolor="#FFFFFF" link="#009966" vlink="#FFCC00" alink="#006666">
<CENTER>
Message sent OK
</BODY>
</HTML>
```