Cika Electrónica

Nota de Aplicación: CAN-004
Título: **Pizarra remota via HTTP con Rabbit 2000 y LCD gráfico (HD61202)**
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.

Como display, utilizaremos un módulo Powertip PG12864, de 128x64 pixels, basado en chips controladores compatibles con el HD61202, de Hitachi, y su clon: el KS0108, de Samsung. No se darán demasiados detalles acerca del display y su software de control, dado que este tema se ha desarrollado en la CAN-003. El lector puede remitirse a dicha nota de aplicación para mayor información.

Hardware

A fin de poder utilizar la interfaz Ethernet en un módulo RCM-2100, utilizaremos un layout diferente al utilizado en la CAN-003. Las conexiones del display quedan como puede apreciarse 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	----------	I/D
PE.3	----------	R/W
PE.0	----------	E
PE.1	----------	CS1
PE.7	----------	CS2

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.

Descripción del funcionamiento

Nuestra pizarra será un display gráfico de 128x64, que mostrará texto en 8 filas de 20 caracteres. La fila superior mostrará la fecha y hora, y los mensajes ingresarán por la fila inferior, desplazando el texto anterior hacia arriba.

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];

	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[21]=0;                          // string <= ancho display
			LCD_scroll();                          // mueve texto anterior arriba
			LCD_printat(7,0,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-003.

```
/* LCD control signals */
#define LCD_CS1        1
#define LCD_CS2        7
#define LCD_E          0
#define LCD_ID  4
#define LCD_RW  3

/* LCD status bits */
#define BUSY   7


void MsDelay ( int iDelay )
{
       unsigned long ul0;
       ul0 = MS_TIMER;              // get current timer value
       while ( MS_TIMER < ul0 + (unsigned long) iDelay );
}


/* Low level functions */

void LCD_SelSide(int side)
{
       if(side)       {
              BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_CS2 );        // Sube CS2
              BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_CS1 );        // Baja CS1
       }
       else    {
              BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_CS1 );        // Sube CS1
              BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_CS2 );        // Baja CS2
       }
}

void LCD_CalcPage(int y,int *page,int *row)
{
       *page=(y>>3);                // page = y/8, parte entera
       *row=y-((*page)<<3);         // row = resto de la división anterior
}

void LCD_Write(int data)
{
       WrPortI ( PADR, &PADRShadow, data );         // escribe datos
       WrPortI ( SPCR, &SPCRShadow, 0x84 );         // PA0-7 = Outputs, datos en el bus
       BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_RW );  // Baja RW (Write)
       BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_E );   // Sube E
       BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_E );   // Baja E
}

int LCD_Read()
{
int data;
       BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_RW );  // Sube RW (Read)
       WrPortI ( SPCR, &SPCRShadow, 0x80 );         // PA0-7 = Inputs, saca datos del bus
       BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_E );   // Sube E
       data=RdPortI ( PADR );                       // Lee datos
       BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_E );   // Baja E
       return(data);
}


int LCD_Status()
{
int status;
```

```
        BitWrPortI ( PEDR, &PEDRShadow, 0,LCD_RS );  // Baja I/D (Cmd)
        do {
                status=LCD_Read();                   // lee status (busy flag)
        } while(status&(1<<BUSY));                   // loop mientras busy=1
        return(status);
}

int LCD_ReadData()
{
        BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_RS );  // Sube I/D (Data)
        return(LCD_Read());
}

void LCD_WriteCmd(int cmd)
{
        LCD_Status();                                // I/D=0 al volver de esta función
        LCD_Write(cmd);
}

void LCD_WriteData(int data)
{
        LCD_Status();
        BitWrPortI ( PEDR, &PEDRShadow, 1,LCD_RS );  // Sube I/D (Data)
        LCD_Write(data);
}
```

La inicialización la haremos diferente esta vez, seteando individualmente los ports. No es estrictamente necesario hacerlo de esta forma:

```
void LCD_init ()
{
        WrPortI ( PEDR,&PEDRShadow,0x00 );           // Make sure Outputs are Low
        BitWrPortI ( PEDDR,&PEDDRShadow,1,LCD_CS1);  // PE.LCD_CS1 = output
        BitWrPortI ( PEDDR,&PEDDRShadow,1,LCD_CS2);  // PE.LCD_CS2 = output
        BitWrPortI ( PEDDR,&PEDDRShadow,1,LCD_E);    // PE.LCD_E = output
        BitWrPortI ( PEDDR,&PEDDRShadow,1,LCD_ID);   // PE.LCD_ID = output

        BitWrPortI ( PEDDR,&PEDDRShadow,1,LCD_RW);   // PE.LCD_RW = output
        BitWrPortI ( PEFR, &PEFRShadow, 0,LCD_CS1 ); // PE.LCD_CS1: no I/O strobe
        BitWrPortI ( PEFR, &PEFRShadow, 0,LCD_CS2 ); // PE.LCD_CS2: no I/O strobe
        BitWrPortI ( PEFR, &PEFRShadow, 0,LCD_E );   // PE.LCD_E: no I/O strobe
        BitWrPortI ( PEFR, &PEFRShadow, 0,LCD_ID );  // PE.LCD_ID: no I/O strobe
        BitWrPortI ( PEFR, &PEFRShadow, 0,LCD_RW );  // PE.LCD_RW: no I/O strobe
        MsDelay ( 1000 );                            // wait for LCD to reset itself

        LCD_SelSide(1);
        LCD_WriteCmd ( '\B00111111' );               // Display on
        LCD_SelSide(0);
        LCD_WriteCmd ( '\B00111111' );               // Display on
}
```

Las funciones de alto nivel son mayormente iguales

```
/* High level functions */

void LCD_home ( void )
{
        LCD_SelSide(1);                              // lado derecho
        LCD_WriteCmd ('\B01000000');                 // address = 0
        LCD_WriteCmd ('\B11000000');                 // display empieza en address 0
        LCD_SelSide(0);                              // lado izquierdo
        LCD_WriteCmd ('\B01000000');                 // address = 0
        LCD_WriteCmd ('\B11000000');                 // display empieza en address 0
}
```

```
void LCD_fill(unsigned char pattern)
{
int i,page;
        LCD_home();
        for(page=0;page<8;page++) {
                LCD_SelSide(0);                         // Controlador izquierdo
                LCD_WriteCmd (0xB8+page);               // Número de página vertical
                for(i=0;i<64;i++)
                        LCD_WriteData(pattern);         // Llena página con datos
                LCD_SelSide(1);                         // Controlador derecho
                LCD_WriteCmd (0xB8+page);               // Número de página vertical
                for(i=0;i<64;i++)
                        LCD_WriteData(pattern);         // Llena página con datos
        }
}

#define LCD_clear() LCD_fill(0)
```

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_scroll1()
{
int i,page,data;

        for(page=2;page<8;page++) {                     // no modifica el renglón superior
                for(i=0;i<64;i++){
                        LCD_WriteCmd (0xB8+page);       // page address
                        LCD_ReadData();                 // dummy read before reading
                        data=LCD_ReadData();            // lee datos en cada renglón
                        if(page==7){
                          LCD_WriteCmd (0x40+i); // set address (contador autoincrementado)
                          LCD_WriteData(0);             // borra último renglón (page 7)
                        }
                        LCD_WriteCmd (0x40+i); // set address (contador autoincrementado)
                        LCD_WriteCmd (0xB7+page);       // apunta a renglón de arriba
                        LCD_WriteData(data);            // y copia los datos
                }
        }
}

void LCD_scroll()
{
        LCD_home();                                             // address 0,0
        LCD_SelSide(1);                                         // derecho
        LCD_scroll1();                                          // scroll
        LCD_SelSide(0);                                         // izquierdo
        LCD_scroll1();                                          // scroll
}
```

El manejo de textos es igual al que hiciéramos en la CAN-003

```
void LCD_putchr ( char chr )
{
const static char font5x7[] = {
<eliminada por cuestiones de espacio, 5 bytes por cada caracter, en formato de pantalla>
};
int i;
        for(i=0;i<5;i++)
                LCD_WriteData(font5x7[5*(chr-0x20)+i]);      // dibuja caracter
        LCD_WriteData(0);                                       // separador (línea en blanco)
}
```

```
void LCD_printat (unsigned int row, unsigned int col, char *ptr)
{
int x;
        do {
                x=(col>9)?6*(col-10):6*col;          // corrije x si col>9
                if(col>9)                            // 0-9 => side=0; 10>20 => side=1
                        LCD_SelSide(1);              // lado derecho, comienza normal
                else {
                        LCD_SelSide(0);              // lado izquierdo
                        x+=4;                        // 4 pixels corrido a la derecha
                }
                LCD_WriteCmd (0xB8+row);             // fila = página
                LCD_WriteCmd (0x40+x);               // address = 6*col ó 6*col-10
                LCD_putchr (*ptr++);                 // escribe caracter
                col++;                               // siguiente columna
        }       while (*ptr);                        // toda la cadena
}
```

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_clear();                                 // borra display

        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(0,0,date);       // muestra en el display
                        LCD_printat(0,12,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>
```