Cika Electrónica

Nota de Aplicación: CAN-018
Título: **Demo Rabbit + Touch Screen + Dynamic C 8**
Autor: Sergio R. Caprile, Senior Engineer

Revisiones	Fecha	Comentarios
0	11/11/03	

Aprovechamos el desarrollo de la nota de aplicación CAN-016 y agregamos funciones con soporte Ethernet, a fin de obtener un demo de las capacidades de manejo de displays gráficos y TCP/IP que nos brinda la combinación Rabbit-Dynamic C 8.

Desarrollaremos un ejemplo de un control sensible al tacto, que muestra el estado de dos salidas. Puede operarse sobre las mismas tanto mediante el panel frontal como via HTTP, a través de una interfaz similar visualizada en forma remota como una página web. Agregamos además dos controles: uno de ellos enviará un email a una direción prefijada, simulando una condición excepcional que debe ser tenida en cuenta por un operador, sin que sea necesario que éste esté observando la pantalla, ni se halle cerca del equipo para escuchar una alarma auditiva. El segundo control, nos permitirá visualizar los emails que el equipo reciba, simulando posibles intervenciones remotas sobre el equipo.

Para soporte del display y la touchscreen utilizaremos las bibliotecas de funciones de Dynamic C 8 y nuestra biblioteca *Cika320240FRST.lib*, con el primitivo hardware de lectura de touch screen.

```
#class auto
// Definimos el hardware de lectura de la touch screen utilizado
#define TSCONTROLLER  1
/* Esto incluye la biblioteca de funciones de Cika que soporta el display
de 320x240 de Powertip */
#use "Cika320240FRST.lib"
```

La configuración de TCP/IP se halla, siguiendo los lineamientos de Dynamic C 8, en el archivo *TCP_CONFIG.LIB*, sito en <Dynamic C path>\LIB\TCPIP. Aprovechamos la oportunidad para demostrar el uso de DHCP.

```
// 1 para IP fija, 5 para DHCP
#define TCPCONFIG 5
```

Definimos los campos que utilizaremos para mandar el email, y la dirección del servidor de correo (SMTP):

```
#define FROM     "RCM2100@cika.com"
#define TO       "soporte@cika.com"
#define SUBJECT  "Rabbit email"
#define BODY     "Alguien presionó el botón...\r\n" \
                                          "Saludos, RCM2100."

#define SMTP_SERVER "cika.com"
```

Luego los datos del servidor POP para poder leer mensajes. La cláusula *POP_PARSE_EXTRA* nos permite acceder en forma detallada a los campos del mensaje:

```
#define POP_HOST       "cika.com"
#define POP_USER       "rcm2100"
#define POP_PASS       "pepito"

#define POP_PARSE_EXTRA
```

Usaremos *xmem*. Incluímos las bibliotecas de funciones de Dynamic C para TCP/IP, HTTP, SMTP y POP3, respectivamente. La biblioteca de DHCP se incluye automáticamente al especificar la configuración.

```
#memmap xmem
#use "dcrtcp.lib"
#use "http.lib"
#use "smtp.lib"
#use "pop3.lib"
```

Importamos las imágenes y páginas que usará el servidor web para mostar la interfaz de control via HTTP. Para almacenarlas, utilizaremos la directiva *#ximport*, que nos permite leer un archivo al momento de compilar y guardarlo en memoria (XMEM), pudiendo referenciarse mediante un puntero:

```
#ximport "ssi.shtml"      index_html
#ximport "rabbit1.gif"    rabbit1_gif
#ximport "ledon.gif"      ledon_gif
#ximport "ledoff.gif"     ledoff_gif
#ximport "button.gif"     button_gif
```

Así, *index_html* es un puntero a la posición de memoria física donde está guardado el archivo *ssi.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.

Importamos los gráficos para los íconos de la interfaz gráfica mediante el display touch screen de Powertip. Los gráficos fueron convertidos mediante la utilidad *fbmcnvtr*, provista con Dynamic C. La misma convierte fonts y bitmaps a formato library. Recordemos que estas bibliotecas, al igual que *Cika320240FRST.lib*, deben estar listadas en el archivo índice de biblioteas de funciones (*LIB.DIR* o equivalente):

```
#use "ledon_bmp.lib"
#use "ledoff_bmp.lib"
#use "button_bmp.lib"
#use "email_bmp.lib"
#use "mail0a_bmp.lib"
```

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 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 a continuación algunas variables globales que utilizaremos más adelante

```
#define BODY_SZ 300
char From[30],Subject[30],Body[BODY_SZ],redirectto[30],led1[15],led2[15];
int msg,n,candraw;
fontInfo fi8x10,fi10x12,fi6x8;
windowFrame bodywin;
unsigned long userX;                          // espacio para estructuras de botones
```

Estas funciones se encargan de actualizar la imagen que representa el estado de las salidas en la pantalla del display. La misma se actualiza cada vez que se opera sobre un control mediante la pantalla del display o a través de la página web. Necesitamos un flag que nos indique que podemos dibujar sobre el display, dado que el mismo puede estar mostrando otra pantalla al momento de llegar el requerimiento via web.

```
void led1show(int state)
{
#GLOBAL_INIT {
       candraw = 1;
}
       if(candraw){                                   // si puede dibujar
         if (state)                                   // muestra la imagen correspondiente
           glXPutBitmap(8, 8, 90,90, ledon_bmp);
         else
           glXPutBitmap(8, 8, 90,90, ledoff_bmp);
       }
}

void led2show(int state)
{
       if(candraw){
         if (state)
           glXPutBitmap(118, 8, 90,90, ledon_bmp);
         else
           glXPutBitmap(118, 8, 90,90, ledoff_bmp);
       }
}
```

Estas funciones se encargan de actualizar la imagen que representa el estado de las salidas en la página web. Son llamadas cada vez que el usuario actúa sobre los controles de la página, como veremos más adelante. La imagen en pantalla se cambia llamando a la función correspondiente, y en la página lo hacemos copiando el nombre de la imagen a enviar y generando un HTTP_REDIRECT, ocasionando que el navegador genere una petición de página, la cual es la misma que la anterior, pero actualizada. La información de redirección está en otra variable, dado que utilizamos DHCP y no conocemos nuestra dirección IP sino hasta el instante de inicialización de la red.

```
int led1toggle(HttpState* state)
{
   if (strcmp(led1,"ledon.gif")==0){                // invierte el estado del LED
      strcpy(led1,"ledoff.gif");
      led1show(0);                                  // y muestra la información actualizada
   }
   else {
      strcpy(led1,"ledon.gif");
      led1show(1);
   }

   cgi_redirectto(state,redirectto);                // informa al navegador para que
   return 0;                                        // actualice la página
}

int led2toggle(HttpState* state)
{
   if (strcmp(led2,"ledon.gif")==0){
      strcpy(led2,"ledoff.gif");
      led2show(0);
   }
```

```
   else {
      strcpy(led2,"ledon.gif");
      led2show(1);
   }

   cgi_redirectto(state,redirectto);
   return 0;
}
```

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 HttpSpec http_flashspec[] =
{
   { HTTPSPEC_FILE,  "/",              index_html,    NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/index.shtml",   index_html,    NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/rabbit1.gif",   rabbit1_gif,   NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/ledon.gif",     ledon_gif,     NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/ledoff.gif",    ledoff_gif,    NULL, 0, NULL, NULL},
   { HTTPSPEC_FILE,  "/button.gif",    button_gif,    NULL, 0, NULL, NULL},

   { HTTPSPEC_VARIABLE, "led1",  0,    led1, PTR16,   "%s", NULL},
   { HTTPSPEC_VARIABLE, "led2",  0,    led2, PTR16,   "%s", NULL},

   { HTTPSPEC_FUNCTION, "/led1tog.cgi",   0, led1toggle, 0, NULL, NULL},
   { HTTPSPEC_FUNCTION, "/led2tog.cgi",   0, led2toggle, 0, NULL, NULL},
};
```

Las primeras seis 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 ssi.shtml
http://MY_IP_ADDRESS/rabbit1.gif -> rabbit1_gif, puntero a rabbit1.gif. Lo mismo ocurre para las demás imágenes en formato 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 dos líneas siguientes definen dos 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 código fuente SHTML de la página, la variable en el programa (que debe ser global), el tipo, y la forma de tratarla para mostrar su valor. En este caso, las variables *led1* y *led2* son strings, comenzando en las posiciones *led1* y *led2* respectivamente.

Las dos últimas líneas asocian los URLs http://MY_IP_ADDRESS/led1tog.cgi y http://MY_IP_ADDRESS/led2tog.cgi con las funciones *led1toggle() y led2toggle()*, respectivamente. Las mismas serán ejecutadas cuando el usuario presione el botón asociado, que tiene un link hacia dicha función, como veremos al analizar el código SHTML. 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, de ser necesario.

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 nueva información.

Escribimos ahora una simple función que nos permita almacenar el último email recibido. Por simplicidad, destruiremos cualquier mensaje anterior, dado que se trata de una demostración. Esta función será llamada para cada línea de texto, cada vez que el cliente POP3 sea ejecutado y compruebe que hay un mensaje disponible, según definiremos más adelante. La cantidad de parámetros de llamada a esta función dependerá de si se ha definido o no *POP_PARSE_EXTRA*.

```
int storemsg(int num, char *to, char *from, char *subject, char *body, int len)
{
static int i;

#GLOBAL_INIT {
       n = -1;
}

       if(n != num) {                              // primera vez: obtiene encabezado
         n = num;
         strncpy(From,from,sizeof(From));
         From[sizeof(From)-1]=0;
         strncpy(Subject,subject,sizeof(Subject));
         Subject[sizeof(Subject)-1]=0;
         Body[0]=0;
         i=0;
       }
       i+=len;
       if((i++)<(BODY_SZ-1)){                      // sucesivas veces: obtiene cuerpo
         strcat(Body,body);
         strcat(Body,"\n");
         Body[sizeof(Body)-1]=0;
         msg=1;
       }
       return 0;
}
```

Llegamos finalmente al cuerpo del programa principal. Luego de la inicialización, procesaremos cada una de las diversas tareas mediante *costates*, aprovechando las funciones de multitarea cooperativo de Dynamic C.

Primero inicializamos el hardware, la biblioteca de funciones gráficas (que inicializa al display) y luego las tipografías a utilizar.

```
void main()
{
int btn,reading,sending,toggle,i,redraw;
char IPaddress[16];
static long POPaddress;

       reading=sending=msg=toggle=0;
       redraw=1;
       pop3_init(storemsg);          // define que se utilice la función “storemsg” para
                                     // recibir mensajes.
       BrdInit();                    // inicializa hardware
       glInit();                     // inicializa gráficos y display

       glXFontInit ( &fi8x10,8,10,0x20,0x7E,Font8x10 );            // define tipografías
       glXFontInit ( &fi10x12,10,12,0x20,0x7E,Terminal9 );
       glXFontInit ( &fi6x8,6,8,0x20,0x7E,Font6x8 );
```

Luego inicializamos la red y el servidor POP. Como utilizamos DHCP, mostramos la dirección IP en la pantalla del display y en la pantalla de Dynamic C para poder acceder a la página web.

```
       glBlankScreen();                                          // borra pantalla
       glPrintf(45,100,&fi10x12,"Inicializando la red...");
       if(sock_init()){
         glPrintf(45,120,&fi10x12,"No inicializa");
         exit(1);
       }
       else {
         inet_ntoa(IPaddress,gethostid());          // traduce IP a texto imprimible
         glPrintf(45,120,&fi10x12,"IP Address = %s",IPaddress);    // muestra en display
         printf("IP Address = %s",IPaddress);                      // y en Dynamic C
         sprintf(redirectto,"http://%s/index.shtml",IPaddress);    //inicializa redirección
       }
```

```
glPrintf(45,140,&fi10x12,"Resolviendo server POP ...");
if(!(POPaddress = resolve(POP_HOST))){                    // resuelve nombre a IP
       glPrintf(45,160,&fi10x12,"No resuelve");
       exit(1);
}
```

Luego inicializamos las estructuras para el despliegue de botones.

```
http_init();                                              // inicializa web server
userX = btnInit( 6 );
btnCreateBitmap(userX,1,0,110,0,1,button_bmp,90,90);      // inicializa botones
btnCreateBitmap(userX,2,110,110,0,1,button_bmp,90,90);    // con gráficos
btnCreateBitmap(userX,3,250,140,0,1,mail0a_bmp,32,32);
btnCreateBitmap(userX,4,220,25,0,2,email_bmp,70,44);
btnCreateText(userX,5,250,200,68,34,1,2,&fi10x12,"Salir");  // botón con texto
btnAttributes(userX,1,0,0,0,0);
btnAttributes(userX,2,0,0,0,0);
btnAttributes(userX,3,0,0,0,0);
btnAttributes(userX,4,0,0,0,0);
btnAttributes(userX,5,0,0,0,0);

strcpy(led1,"ledon.gif");                   // imágenes web y salidas:
strcpy(led2,"ledoff.gif");                  // inicializa estado por defecto
```

Y finalmente el loop principal. Las tareas definidas son:

- ➔ TCP/IP: procesamiento de HTTP y de SMTP y/o POP3 cuando hay un mensaje pendiente.
- ➔ Botones: espera que se presione algún botón y ejecuta la tarea correspondiente.
- ➔ POP3: espera unos dos minutos y luego interroga al servidor si tiene un mensaje. Si es así, lo baja.

Cada tarea entrega el controla a las demás cuando no tiene nada que hacer.

```
while (1) {

  if(redraw){                               // Muestra pantalla de control al inicio
    redraw=0;                               // o luego de lectura de email
    glBlankScreen();
    while (!btnDisplayLevel(userX,1));      // Muestra botones de nivel 1
    led1show(strcmp(led1,"ledon.gif")==0);  // Muestra estado salidas
    led2show(strcmp(led2,"ledon.gif")==0);
    glPrintf(4,230,&fi6x8,"Demo Seminario Rabbit, Cika Electronica S.R.L., 2003");
  }

  costate {                                 // TCP/IP
    http_handler();                         // atiende página web
    if(sending)
        if (smtp_mailtick()!=SMTP_PENDING)  // atiende SMTP (envío emails)
          sending=0;
    if(reading)
        if(pop3_tick()!= POP_PENDING)       // atiende POP3 (recepción emails)
          reading=0;
    if(msg){
        btnDisplay(userX,4);                // muestra botón si corresponde
        msg=0;                              // (hay mensaje para leer)
     }
  }
  costate {                                 // procesa botones
    waitfor (( btn = btnGet(userX) ) >= 0 ); // espera botón apretado
        switch (btn){
          case 1:
            led1toggle(NULL);               // opera sobre salida 1
            break;
          case 2:
            led2toggle(NULL);               // opera sobre salida 2
            break;
          case 3:
```

```
                smtp_sendmail(TO, FROM, SUBJECT, BODY);  // comienza envío de email
                sending=1;                       // habilita smtp_mailtick()
                break;
              case 4:
                n=-1;                                    // muestra email
                candraw=0;                       // impide actualización de pantalla
                while (!btnClearLevel(userX,1));         // borra botones de nivel 1
                btnClear(userX,4);                       // borra botón de nuevo mensaje
                glBlankScreen();                         // borra pantalla
                glPrintf(4,4,&fi8x10,"From: %s",From);   // Encabezado
                glPrintf(4,22,&fi8x10,"Subject: %s",Subject);
                glSetBrushType(PIXXOR);                  // Bloque en video inverso
                glBlock(0,0,320,36);
                glSetBrushType(PIXBLACK);                // Cuadro de texto
                TextWindowFrame(&bodywin,&fi10x12,0,38,320,202);
                TextBorderInit(&bodywin,SINGLE_LINE,"");
                TextBorder(&bodywin);
                TextPrintf(&bodywin,Body);               // Cuerpo del mensaje
                btnDisplay(userX,5);                     // Botón para salir
                break;
              case 5:
                candraw=redraw=1;                        // Sale, permite mostrar
                btnClear(userX,5);                       // la pantalla principal
                break;
            }
        }
        costate {                                        // Consulta si hay mensajes
          waitfor(DelaySec(120));                        // cada 2 minutos
          pop3_getmail(POP_USER, POP_PASS, POPaddress);
          reading=1;                                     // llamando a pop3_tick()
        }
      }
}
```

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. Si el parámetro es *DelaySec(n)*, el resultado concreto es que a cada paso por esa instrucción, se devuelve el control a las demás tareas hasta tanto hayan transcurrido *n* segundos, 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 SHTML que será procesada por el servidor HTTP (provisto por Dynamic C), sin intervención de nuestra parte.

Veamos *ssi.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 W3 HTML//EN">
<HTML>
<HEAD>
<TITLE>Demo</TITLE>
</HEAD>

<BODY topmargin="0" leftmargin="0" marginwidth="0" marginheight="0"
       bgcolor="#FFFFFF" link="#009966" vlink="#FFCC00" alink="#006666">
```

Muestra el logo de Rabbit

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

Define las variables de salida (SSI) y los links a los CGI, dentro de una tabla por razones estéticas.

```
<TABLE BORDER="0" CELLSPACING="2" CELLPADDING="1">
       <TR>
              <TD> <img SRC="<!--#echo var="led1"-->"> </TD>
```

```
                <TD> <img SRC="<!--#echo var="led2"-->"> </TD>
        </TR>
        <TR>
                <TD> <A HREF="/led1tog.cgi"> <img SRC="button.gif"> </A> </TD>
                <TD> <A HREF="/led2tog.cgi"> <img SRC="button.gif"> </A> </TD>
        </TR>
</TABLE>
</BODY>
</HTML>
```

El servidor detecta `<!--#echo var="led1"-->` y lo reemplaza por el contenido de la variable *`led1`*.
Al hacer click sobre la imagen *button.gif*, el navegador pide */led1tog.cgi* o */led2tog.cgi*, según corresponda.