Revisión	Fecha	Comentario	Autor
0	15/04/2009		Ulises Bigliati

ÍNDICE

Introducción

En esta oportunidad vamos a intentar aportar una herramienta relativamente nueva a los desarrolladores de sistemas embebidos, particularmente a quienes habitualmente incorporan funcionalidad basada en web a sus trabajos. Se trata de un método de programación que establece una forma diferente de utilizar los estándares ya existentes para lograr aplicaciones basadas en web mas eficientes.
Esta técnica de programación llamada AJAX (**A**synchronous **J**avaScript **A**nd **X**ML) fue popularizada por Google en 2005, con *Google suggest.* En esta nota proponemos su implementación en las páginas web servidas por Rabbit, ya que AJAX es totalmente independiente del servidor HTTP.

Conceptos necesarios

Para comprender los conceptos que serán abordados se requieren conocimientos de HTML y JavaScript. Adicionalmente será deseable haber tenido al menos una aproximación a RabbitWeb, que es el lenguaje de scripting del lado del servidor que provee Rabbit, o en su defecto, conocer algún otro lenguaje de scripting del lado servidor como puede ser PHP, ya que conceptualmente es muy similar. Por otra parte, sería conveniente al menos haber visto alguna vez algún archivo en formato XML.

Objetivos

- Exponer las bases que permiten entender y utilizar la técnica de programación propuesta por AJAX.
- Realizar una sencilla aplicación que implemente AJAX sobre el entorno de desarrollo de Rabbit.

Fuera de alcance

No es objeto de esta nota brindar una descripción exhaustiva sobre AJAX ni tampoco ahondar en detalles de los estándares subyacentes; si se desea acceder a un tutorial completo puede obtenerse, por ejemplo, en esta dirección: http://www.w3schools.com/Ajax/Default.Asp

La aplicación propuesta

Para la realización de este ejemplo utilizamos el nuevo módulo MiniCore de Rabbit, el RCM5700 junto con Dynamic C 10.46 y el módulo de software RabbitWeb.
El ejemplo podrá ser compilado sobre otros módulos Rabbit junto con su versión correspondiente de Dynamic C sin prácticamente efectuar modificaciones.

Funcionamiento del programa

Consiste en el monitoreo en tiempo real vía web de los valores de 8 canales analógico simulados y de la fecha y hora presentes en el módulo Rabbit. Por otra parte también es posible sincronizar la hora de la PC con el RTC del módulo. Los valores cambian aleatoriamente cada 500 ms y desde la página web vamos a ver la actualización cada segundo sin sufrir ninguna recarga de la página visualizada. Nótese en la imagen que la página está cargada con varios archivos gráficos, incluso, sobre uno de ellos se imprimen los valores que extraemos del equipo remoto, hemos hecho esto intencionalmente para remarcar la diferencia al utilizar AJAX en una aplicación web. Como verán al ejecutar la demo, la ventaja es notable con respecto al método tradicional.

AJAX (Asynchronous JavaScript And XML)

Que es?

Para comenzar con la presentación de AJAX, debemos insistir en que no se trata de algo nuevo, sino mas bien de una forma, si se quiere novedosa, de utilizar los estándares ya existentes para la programación en entorno Web. AJAX está basado en los siguientes estándares:

- JavaScript
- XML
- HTML
- CSS

En esencia, AJAX es una combinación de código JavaScript y HTTP request's, más precisamente, se trata de utilizar un objeto (una pieza de software) que es ejecutado en segundo plano por el browser, con el cual se puede interactuar mediante JavaScript permitiéndonos realizar HTTP requests en forma asíncrona, de tal forma, es posible actualizar datos en una página Web sin necesidad de recargar la página. El objeto que constituye el corazón de esta técnica se llama **XMLHttpRequest**.
Es importante destacar que AJAX es una tecnología del explorador web, independiente de los servidores HTTP, por ello es perfectamente realizable su utilización inmediata junto con Rabbit.

Para que sirve?

Tradicionalmente cuando se solicita un recurso a un servidor web o se le envía información, se utilizan formularios HTML mediante métodos GET o POST, efectuando previamente una operación "`submit`" sobre el formulario en cuestión. Luego se debe esperar a que el servidor responda con una nueva página completa como resultado.

Dado que este esquema requiere que el servidor envíe una página completa por cada solicitud o cada vez que requerimos actualizar datos (ya sea en forma manual o automática) las aplicaciones web tradicionales tienden a ser lentas, poco amigables y poco eficientes en cuanto a que se transmite información redundante innecesariamente.

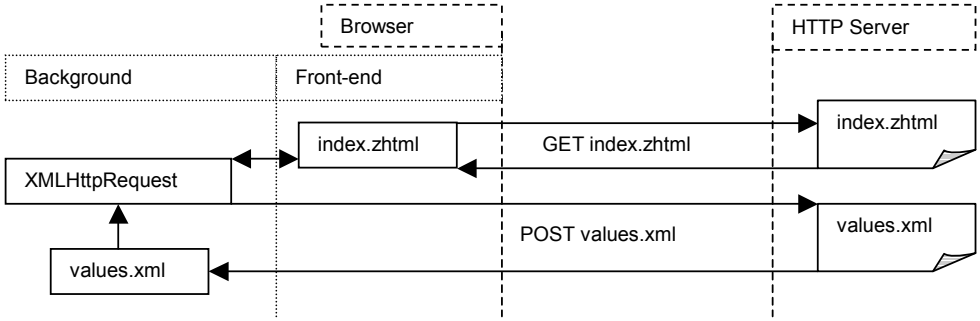
Con AJAX, nuestro código JavaScript puede solicitar/enviar al servidor solo los datos que son necesarios a través del objeto **XMLHttpRequest** y luego actualizar la página web que el usuario tiene en pantalla en segundo plano y sin necesidad de recargarla, concretándose un mínimo intercambio de datos entre el browser y el servidor.

En otras palabras, el objeto XMLHttpRequest, mediante requerimientos HTTP (GET/POST) puede actualizar una página web luego de que esta ha sido cargada.

Un ejemplo de ello es *Google Suggest* : al escribir sobre la caja de texto del buscador se envía la palabra que vamos escribiendo al servidor, quien devuelve una lista de sugerencias de búsqueda, la cual se muestra dinámicamente sobre la página original, sin recargarla en absoluto.

Cómo funciona?

El siguiente esquema intenta graficar en forma simple el modo de funcionamiento del mecanismo propuesto por AJAX. El concepto es muy sencillo y podría decirse que está basado en la separación de los datos de la presentación y en la solicitud y recepción del archivo de datos resultante en segundo plano y en forma asíncrona:



Cuando un usuario solicita la página `index.zhtml` el servidor la concede, y una vez recibida por el browser se ejecuta una función Javascript contenida en aquella página conforme suceda algún evento. Esta función crea el objeto `XMLHttpRequest` y le indica básicamente cuál es la página de datos que debe solicitar (`values.xml`), y cómo asignar los datos una vez recibidos. Una vez hecho esto, la actualización se realiza en segundo plano, sin recargar nunca más la página `index.zhtml`, solo se solicita `values.xml` todas las veces que sea necesario, efectuando una mínima transfertencia de datos y evitando las molestas recargas de páginas. En este contexto cabe remarcar lo siguiente:

1)En el servidor web se deben separar los datos variables de los objetos estáticos. Es decir: el front-end estático conteniendo imágenes, logos, texto fijo, tablas, etc, iría en un archivo (*.zhtml* si contiene código *RabbitWeb* como es nuestro caso). Y los datos variables, irían en otro archivo *(.zhtml*, *.php*, *.xml*) ya sea en texto plano, o formateado según nos convenga. Los archivos XML son los que mejor se adaptan para manejar múltiples variables, así que los vamos a utilizar en nuestra demo.

2) El front-end debe contener el código en Javascript necesario para crear el objeto fundamental de AJAX, establecer los valores de sus propiedades y reaccionar a la recepción del archivo de datos, asignando los valores a los objetos HTML correspondientes. Normalmente todo esto se realiza en una misma función de Javascript.

3) Todo este mecanismo se diseña, codifica y ejecuta en el explorador web, del lado del cliente, es decir se trabaja sobre las páginas web (*.html*, *.zhtml*, *.xml*). El servidor HTTP, que en este caso se ejecuta en nuestro módulo Rabbit, jamás se entera ni le interesa nada de todo esto.

Cómo se usa?

Tal como puede observarse, la clave de AJAX es el objeto *XMLHttpRequest*, y este objeto es soportado por la mayoría de los exploradores web: *Internet Explorer 5.0+, Safari 1.2, Mozilla 1.0 / Firefox, Opera 8+,* y *Netscape 7.*
Para utilizarlo, y así sacar provecho de las ventajas de AJAX simplemente hay que instanciar este objeto en la sección de código JavaScript de nuestras páginas web y utilizarlo convenientemente, según veremos mas adelante.
Sin embargo si queremos que nuestra aplicación funcione en todos los navegadores, debemos tener en cuenta que salvo *Internet Explorer*, el resto de los navegadores del Universo utilizan el objeto XMLHttpRequest en forma nativa, es decir, usan el estándard y la instanciación del objeto se realiza:

```
// Firefox, Opera 8.0+, Safari
xmlHttp = new XMLHttpRequest();
```

Los salmónidos navegadores de Microsoft, por el contrario, lo instancian como objetos ActiveX, pero esto no es todo, sino que además:

```
//IE >= 6.0
xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");

//5.5 <= IE < 6.0
xmlHttp=new ActiveXObject("Microsoft.XMLHTTP");
```

Es decir, en nuestro código debemos evaluar si la versión del *IE* es inferior o superior a la 6.0, pues la forma de instanciar el objeto que necesitamos cambia conforme a esto.
En definitiva, para obtener una instancia de la clase `XMLHttpRequest` en la práctica vamos a escribir de manera genérica el siguiente código[1]:

[1] La sentencias try/catch http://www.w3schools.com/jS/js_try_catch.asp

```
<head>
<script language="javascript" type="text/javascript">
        function ajaxFunction()
        {
                var xmlHttp;

                try
                {       // All browsers of the Universe
                        xmlHttp = new XMLHttpRequest();
                }
                catch (e)
                {       // Internet Explorer
                        try
                        {
                           xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
                        }
                        catch (e)
                        {
                           try
                           {
                              xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
                           }
                           catch (e)
                           {
                               alert("Your browser does not support AJAX!");
                               return false;
                           }
                        }
                }
        }
</script>
</head>
```

Con este código ya tenemos una instancia del objeto llamada `xmlHttp`, que podremos utilizar en el ámbito de la función que hemos llamado `ajaxFunction()` luego nos queda utilizar las propiedades y métodos del objeto de la forma adecuada.

Propiedad *onreadystatechange*

En esta propiedad almacenamos el código de una función que será la responsable de procesar la respuesta del servidor web al cual le estemos solicitando los datos. Con el siguiente código realizamos la asignación que parece algo extraña, pero sin embargo equivale a pasarle al objeto `xmlHttp` un puntero a la función:

```
xmlHttp.onreadystatechange    =
                function()
                {
                   //Nuestro código para procesar los datos
                   //que esperamos del servidor
                }
```

Propiedad *readyState*

La propiedad `readyState` guarda el estado de la respuesta del servidor, y cada vez que esta cambia la función que asignamos a la propiedad `onreadystatechange` se ejecuta. Los siguientes son los valores posibles de esta propiedad:

Estado	Descripción
0	Request no inicializado
1	Request inicializado
2	Request enviado
3	Request en proceso
4	Request completado

Particularmente nos va interesar saber cuando esta propiedad toma el valor de cuatro, es decir, cuando podemos disponer de los datos, así que normalmente el código anterior iría tomando esta forma:

```
xmlHttp.onreadystatechange    =
                function()
                {
                  if(xmlHttp.readyState == 4)
                  {
                   //Nuestro código para procesar los datos
                   //que esperamos del servidor
                  }
                }
```

Propiedad `responseText`

Los datos enviados por el servidor pueden obtenerse como un string desde esta propiedad. Por ejemplo, en nuestra función agregaríamos el código correspondiente para que el resultado sea este:

```
xmlHttp.onreadystatechange    =
                function()
                {
                  if(xmlHttp.readyState == 4)
                  {
                        alert(xmlHttp.responseText);
                  }
                }
```

Propiedad `responseXML`

En forma análoga a la propiedad anterior, `responseXML` devuelve la respuesta del servidor en formato XML, es decir, devuelve un objeto que representa un documento XML, el cual puede ser explorado y parseado usando el modelos de objetos DOM[2] de la W3C. Esto es bastante más complicado, pero es lo que usaremos ya que ofrece las mayores prestaciones, el documento XML podría obtenerse entonces, así:

```
        var xmlDoc = xmlHttp.responseXML;
```

Sin embargo, en nuestra práctica lo haremos de esta forma:

```
        var xmlDoc = xmlHttp.responseXML.documentElement.getElementsByTagName("ch");
```

porque esto obedece a la naturaleza del archivo XML que estamos esperando, pero por ahora no nos detendremos en esto ya que nos ocuparemos mas adelante, solo vamos a decir que luego de esa línea podremos disponer de un listado de valores formateados de esta manera: <ch>[valor]</ch> que serán accesibles mediante el objeto `xmlDoc`. Veamos entonces como va quedando nuestra función:

```
function ajaxFunction()
{
        var xmlHttp;
        try{
        //catch{...etc...
        xmlHttp.onreadystatechange    =
                function()
                {
                  if(xmlHttp.readyState == 4)
                  {
                        var xmlDoc = xmlHttp.responseXML.documentElement.getElementsByTagName("ch");
                        //A continuación se debe procesar el documento XML mediante xmlDoc
                        //asignando los valores obtenidos a nuestra página HTML estática.
                  }
                }
}
```

[2] DOM (Document Object Model) El documento XML se estructura en forma de árbol, y se lo recorre a través de sus nodos.Ver http://www.w3schools.com/xml/xml_dom.asp

Método `open()`

El método tiene la forma `open("method","URL",async,"uname","pswd");`
- `method`: los valores que acepta son "`GET`", "`POST`", or "`PUT`"
- `URL`: El recurso que queremos solicitar (puede ser relativo o absoluto).
- `async`: especifíca cómo se manejará el requerimiento, acepta los valores: `true` (forma asíncrona), `false` (forma síncrona).
-`uname,pswd`: son opcionales.

En nuestra demostración utilizamos el método "`POST`", pedimos al servidor el archivo "`values.xml`" y manejamos el request en forma asíncrona:

```
xmlHttp.open("POST","values.xml",true);
```

Método `send()`

Este método simplemente envía el requerimiento HTTP, y tiene la forma `send(content)`; donde el valor de `content` dependerá del parámetro `method` usado en `open()`. Si se utilizó `GET` el parámetro valdrá `null` si se utilizó `POST`, `content` será un string con los parámetros enviados en el requerimiento HTTP y tendrá la forma “param1=valor1&paramN=valorN”.
Nosotros construímos el valor de esta cadena de texto de esta forma:

```
var params = (syncTime)? "strDateTime=" + getTime(): "timeRefresh=1";
```

Lo cual obedece a un aspecto de diseño particular de la aplicación que utiliza RabbitWeb que corre en el Rabbit, lo importante aquí es que enviaremos un solo parámetro mediante el método POST, este: `“strDateTime=dd/mm/aaaa hh:m:ss”` o este otro `“timeRefresh=1”` en forma excluyente. Luego, obviamente realizamos el envío: `send(params);`

Método `setRequestHeader ()`

Este método es necesario cuando se utiliza el método `POST` y sirve para añadir los encabezados del requerimiento HTTP, tiene la forma `setRequestHeader("label","value")`. En nuestra aplicación lo utilizamos así:

```
xmlHttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xmlHttp.setRequestHeader("Content-length", params.length);
```

Donde `params.length` es la longitud total del HTTP request enviado con el método `POST`.

De vuelta a la función AJAX

Finalmente después de tanta cháchara llega el momento de poner juntos a todos los ingredientes que venimos analizando, aunque todavía queda algo por explicar que por claridad abordaremos más adelante y es la forma en que exploramos y utilizamos el documento XML que le pedimos al servidor con el método `open()`.
Lo que sigue es el código completo de nuestra función que utiliza AJAX y que hace todo el trabajo que estamos buscando:

```
<head>
<script language="javascript" type="text/javascript">
        function ajaxFunction(syncTime)
        {
                var xmlHttp;

                try
                {       // All browsers of the Universe
                        xmlHttp = new XMLHttpRequest();
                }
                catch (e)
                {       // Internet Explorer
                        try
                        {
                           xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
                        }
                        catch (e)
                        {
                           try
                           {
                              xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
                           }
                           catch (e)
                           {
                               alert("Your browser does not support AJAX!");
                               return false;
                           }
                        }
                }
                xmlHttp.onreadystatechange    =
                function()
                {
                  if(xmlHttp.readyState==4)
                  {
                    var myValues = document.getElementsByName("analog");
                    var xmlDoc = xmlHttp.responseXML.documentElement.getElementsByTagName("ch");
                    for(var i=0;i<=7;i++)
                       myValues[i].innerHTML = xmlDoc[i].childNodes[0].nodeValue;

                    xmlDoc = xmlHttp.responseXML.documentElement.getElementsByTagName("datetime");
                    document.getElementsByName("datetime")[0].innerHTML  =
                                                          xmlDoc[0].childNodes[0].nodeValue;

                  }
                }
                xmlHttp.open("POST","values.xml",true);
                var params = (syncTime)? "strDateTime=" + getTime(): "timeRefresh=1";
                xmlHttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
                xmlHttp.setRequestHeader("Content-length", params.length);
                xmlHttp.send(params);

                timer=setTimeout('ajaxFunction()',1000);
        }
</script>
</head>
```

Cabe destacar el uso del parámetro `syncTime`, destinado a distinguir entre dos formas de uso que le podemos dar a la función desde la página principal, a saber: 1) Solo actualización de datos o 2) Puesta en hora del RTC y posterior actualización de datos. La diferencia reside en la forma en que armamos el requerimiento HTTP en la variable `params`. Según el parámetro enviado, forzamos al servidor a ejecutar la tarea que corresponda en cada el caso.
En cuanto a la última línea, su motivación quizá pueda deducirse fácilmente, pero de todas maneras la explicamos bajo el siguiente subtítulo.

Donde, cómo y cuando?

Hasta acá parece estar todo muy lindo, pero el lector perspicaz sabe que falta establecer, desde donde, como y cuando invocamos a aquella función. Despejamos aquí los interrogantes.
En un esquema tradicional usaríamos formularios y el método `submit` para envíar y solicitar datos al servidor, con AJAX esto no es así, ya que el envío y recepción de datos los maneja una función Javascript

como la que acabamos de describir, y esta función es invocada con subordinación al suceso de algún evento que hayamos programado (click en un botón, tipeo de una letra sobre una caja de texto, timers, etc). Bien, en nuestra página, esto lo tenemos que hacer de dos maneras (o quizá de tres):

1. Definimos un botón y en su evento `onclick` realizamos la llamada a la función `ajaxFunction(syncTime)`, nótese que le pasamos "1" como valor para el argumento, es decir, con este llamado queremos sincronizar la hora del RTC del Rabbit con nuestra PC, el HTML luce así:

```
<input type="button" value="SyncTime" onclick="ajaxFunction(1);">
```

2. Invocamos la función en el evento `onload` del `body`, es decir, cuando se carga la página, aquí le pasamos "0" o `false` como argumento, y dice así:

```
<body bgColor="#ffffff" onload="ajaxFunction(0);">
```

3. Creamos un timer al final de la función AJAX, de tal forma que nos aseguramos la actualización automática permanente cada 1000 ms, así, la primera vez, llegamos a la función gracias al evento `onload` del `body`, y a partir de allí, podría decirse que la función se llama a sí misma en forma recurrente con la colaboración del timer. Recordar que esta es la última línea dentro de la función `ajaxFunction()`:

```
timer=setTimeout('ajaxFunction()',1000);
```
[3]

La parte estática de este asunto

Habiendo repasado todo lo que va y viene, nos queda todavía entre otras cosas, darle un vistazo a las estanterías. Esto es, el resto de los objetos HTML que son referidos por el código Javascript que acabamos de describir, esto es algo mas sencillo, pues no hay código ejecutable que explicar.

La página principal `index.zhtml`

Para continuar poniendo el foco en la claridad, no vamos a desplegar aquí la página entera por razones obvias, ya que además, puede verse en el proyecto, en cambio haremos un recorte de las secciónes mas trascendentes.
Veamos la sección para los datos analógicos que estaríamos monitoreando en forma permanente y en tiempo real y que son los que nuestra función AJAX solicita cada un segundo, tenemos pues, una tabla con espacio para 8 registros, en cada registro hay una celda con un elemento **`<a name="analog"></a>`** en total, ocho de estos elementos, virtualmente uno por cada canal analógico. Acá hay algo importante, que permite entender la forma de asignación de los valores que llegan vía XML: los ocho elementos tienen el mismo nombre, por lo tanto, conforme al modelo de objetos DOM del W3C, mediante el método `document.getElementsByName("analog")`, podemos obtener fácilmente un array de 8 elementos `<a>` que será fácilmente manipulado en Javascript en cualquier momento para realizar cualquier modificación de sus propiedades, lo cual se vería inmediatamente reflejado en pantalla.

```
<table  valign="top" class="data" >
       <tr>
              <th align="middle" class="data"><a>Entrada Analogica</a> </th>
              <th align="middle" class="data"><a>Valor</a></th>
              <th align="middle" class="data"><a>Unidad</a></th>
       </tr>
       <tr>
              <td class="data"><a>An0</a></td>
              <td class="data"><a name="analog"></a></td>
              <td class="data"><a>mV</a></td>
       </tr>
       ...
       ...
       <tr>
             <td class="data"><a>An7</a></td>
             <td class="data"><a name="analog"></a></td>
             <td class="data"><a>mV</a></td>
       </tr>
</table>
```

[3] Ver uso de timers en Javascript: http://www.w3schools.com/js/js_timing.asp

El mismo concepto descripto arriba es aplicado al dato variable restante, que muestra la fecha y hora presentes en el RTC del módulo Rabbit, aquí el elemento es **`<a class="foot" name="datetime"></a>`** y desde el código Javascript será accedido así `document.getElementsByName("datetime")`

```
<table name="status"  class="data">
       <tr>
              <th class="foot"><a>Fecha y hora del equipo:</a></th>
              <td class="foot"><a class="foot" name="datetime"></a></td>
              <td class="foot">
                     <input type="button" value="SyncTime" onclick="ajaxFunction(1);">
              </td>
       </tr>
</table>
```

Para finalizar, véase también la ubicación del botón de puesta en hora, cabe destacar que hasta aquí no explicamos cómo obtener la hora actual de la PC, ya que se trata de un tema accesorio, de todas formas, como puede verse en el código, la función AJAX invoca a la función *getTime()* que está definida en el archivo ZHTML y que devuelve un string con el formato adecuado para ser enviado como parámetro del requerimiento HTTP, dicho parámetro es un string con formato "dd/mm/aaaa hh:mm:ss" que es recibido por el servidor web en el Rabbit, y es utilizado, función correspondiente mediante, para modificar la fecha y hora del RTC.

El archivo de datos `values.xml`

Para terminar de describir el mecanismo que estamos utilizando, nos queda hablar sobre el importante archivo `values.xml`; por supuesto, queda fuera del alcance de esta nota el abordaje de un estándard tan amplio como XML, pero como siempre, el lector puede acudir a las referencias[4] para ampliar lo presente. De todas formas, vamos a hacer una breve explicación de este archivo, que está conformado como puede verse a continuación:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<values>
       <analog>
              <ch> <?z printf("%.2f",@ad_inputs[0]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[1]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[2]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[3]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[4]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[5]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[6]) ?> </ch>
              <ch> <?z printf("%.2f",@ad_inputs[7]) ?> </ch>
       </analog>
       <datetime> <?z print(@strDateTime) ?> </datetime>
</values>
```

Pasamos a explicarlo:
La primer línea es la declaración de XML, y define la versión y codificación de caracteres utilizada.
La línea siguiente, es el elemento raíz del documento: `<values></values>` y nos da una breve idea sobre el contenido.
Los elementos en un archivo XML se estructuran como un árbol, que tiene inicio en el elemento raíz y se ramifica hacia sus elementos hijos, este caso: `<analog></analog>` y `<datetime></datetime>` que a su vez pueden tener sus propios elementos hijos: `<ch></ch>` en el caso del elemento `<analog>` o que pueden presentar contenido de texto, como en el caso de los elementos `<ch></ch>` y `<datetime></datetime>` que contienen el valor final del dato, que en definitiva es lo que nos interesa transportar. Remarcamos esta palabra, porque uno de los usos más importantes de XML es justamente el transporte de datos.

[4] http://www.w3schools.com/xml/default.asp

El archivo XML en el servidor

En definitiva este archivo nos sirve para transportar una estructura de datos como la siguiente:

values
- analog
  - ch(0)
  - ch(1)
  - ch(2)
  - ch(3)
  - ch(4)
  - ch(5)
  - ch(6)
  - ch(7)
- datetime

el tratamiento dinámico de la obtención de los datos lo logramos gracias a la combinación del archivo XML con RabbitWeb, por eso, en cada lugar donde debemos colocar un dato extraído del programa que corre en el módulo Rabbit vemos una sentencia similar a la que sigue:

```
<?z printf("%.2f",@ad_inputs[0]) ?>
```

Esa línea, le dice al *zhtml_handler* que debe imprimir el contenido de la variable `ad_inputs[0]` en su lugar. RabbitWeb[5] reconoce que debe hacer esto "al vuelo" mientras el servidor web entrega cada línea del archivo al cliente (browser) que pide esta página y reconoce las líneas que se deben procesar dinámicamente del lado servidor mediante los tags `<?z ?>`.

Al servidor HTTP de Rabbit, debemos indicarle que cuando un cliente solicite un archivo ".XML", debe procesarlo con el *zhtml_handler.* Las siguientes líneas de código son todo lo necesario para tener este archivo en el servidor, listo para ser entregado al browser:

```
#ximport "IWeb\values.xml"    values_xml //importar el archivo en memoria flash y obtener un puntero
                                         //a su ubicación (values_xml).
SSPEC_MIMETABLE_START
...
   SSPEC_MIME_FUNC(".xml", "text/xml", zhtml_handler), //procesar archivos ".xml" con zhtml_handler
...
SSPEC_MIMETABLE_END
SSPEC_RESOURCETABLE_START
...
       SSPEC_RESOURCE_XMEMFILE("/values.xml", values_xml),//values.xml está en la dirección indicada
...                                                       //por el puntero values_xml
SSPEC_RESOURCETABLE_END
```

El archivo XML del lado del cliente

Una vez que ya tenemos claro, cómo está estructurado el archivo de datos, y cómo este se genera del lado del servidor, veamos cómo se procesa una vez recibido del lado del browser, recordemos que esto lo hacemos dentro de la función AJAX, que para asignar los datos se moverá dentro de la página *zhtml* de acuerdo al modelo DOM (document object model) de la W3C, es por eso que utilizamos ciertos métodos (funciones) de nombres larguísimos y odiosos, que están implementados en los objetos que conforman este modelo, y todo ello está presente en forma nativa en todos los navegadores web:

Primero, obtenemos un array de los elementos "contenedores" de los datos, como vimos, se nos había ocurrido meter a estos en una tabla, y elegimos elementos `<a>` y les pusimos `name="analog"` (ver pag. 9) Entonces, utilizamos el método `getElementsByName` del objeto `document`[6] (que representa a toda la página html) y le pasamos el nombre de los elementos que queremos reunir (`"analog"`):

```
var myValues = document.getElementsByName("analog");
```

Ahora ya tenemos a todos los elementos accesibles dentro del array `myValues`.
Luego hacemos algo muy parecido, pero con el archivo XML que recibimos, recordar que está almacenado en la propiedad `responseXML` del objeto `XMLhttp`, aunque más precisamente, está en un objeto de tipo `document`, que está incluido en la propiedad citada, entonces ahora usamos un método parecido, pero que se llama `getElementsByTagName`, que busca elementos según su nombre (diferente a buscar elementos según el valor de su propiedad `name`), entonces le pedimos todos los `<ch>` (ver página anterior):

```
var xmlDoc = xmlHttp.responseXML.documentElement.getElementsByTagName("ch");
```

Y ahora que ya tenemos todos los elementos `<ch>` en un array que llamamos `xmlDoc` podemos realizar las asignaciones. Cada `myValues[i]` es un elemento `<a>`, y `.innerHTML` es una propiedad que representa "lo que va entre `<a>` y `</a>`". Del otro lado de la asignación, cada `xmlDoc[i]` es un elemento `<ch>`, y `.childNodes[0]` referencia al primer nodo hijo disponible, que como sabemos, el único hijo que tiene este elemento es el valor final que estamos buscando, así es que lo obtenemos con la propiedad `.nodeValue` :

[5] http://www.rabbit.com/documentation/docs/manuals/TCPIP/UsersManualV2/ (Sección 5 RabbitWeb)
[6] http://www.w3schools.com/HTMLDOM/dom_obj_document.asp

```
for(var i=0;i<=7;i++)
       myValues[i].innerHTML = xmlDoc[i].childNodes[0].nodeValue;
```

Hasta acá ya tenemos los valores actualizados en nuestra página, rápido y sin pestañar. Nos queda buscar la fecha y hora actuales con los mismos conceptos:

```
xmlDoc = xmlHttp.responseXML.documentElement.getElementsByTagName("datetime");
```

Y asignar la fecha y hora también con los mismos conceptos:

```
document.getElementsByName("datetime")[0].innerHTML  = xmlDoc[0].childNodes[0].nodeValue;
```

Ahora ya podemos descansar un poco porque lo peor (o lo mejor) ya está hecho.

Estructura de archivos del proyecto

Físicamente esta demostración está organizada según puede apreciarse en el siguiente esquema:
De los archivos **index.zhtml** y **values.xml** ya hablamos bastante, de **datetime.lib** no vamos a hablar porque solo tiene un par de funciones accesorias (`get_time` / `set_time`), pero sobre **iweb.lib** y **CoAN-a10.c** vamos a hacer un breve repaso enseguida para terminar de cerrar algunos aspectos de la aplicación.

myProyects
- |___CoAN-010
  - |___**CoAN-010.c**
  - |___CoAN-010.dcp
  - |___lib.dir
  - |___myLibs
    - |___**iweb.lib**
    - |___**datetime.lib**
  - |___iWeb
    - |___**index.zhtml**
    - |___**values.xml**
    - |___style.css
    - |___rabbit.jpg
    - |___conti.gif
    - |___r4k.jpg

El archivo de programa

El archivo que contiene el `main()` de esta aplicación, prácticamente no aporta nada al asunto que venimos discutiendo, mejor dicho, no aporta nada. Solo debemos nombrar un par de puntos destacables:

1) La sección de configuración de red, que es lo único que el usuario debería configurar para correr el programa:

```
[...]
/*************************************************
*               NETWORK CONFIGURATION                                    *
*************************************************/
#define TCPCONFIG 1
#define _PRIMARY_STATIC_IP  "192.168.10.55"
#define _PRIMARY_NETMASK    "255.255.255.0"
#define MY_GATEWAY          "192.168.10.1"
```

2)La presencia del array global que contiene los famosos valores que transferimos vía HTTP en el archivo XML hacia el browser del cliente.

```
[...]
float ad_inputs[8];
[...]
```

(Recuerden líneas como esta en el archivo XML: `<ch> <?z printf("%.2f",@ad_inputs[0]) ?> </ch>` *)*

Y la forma vergonzosa en que estos valores son simulados:

```
[...]
      /*********************************************************************
       * Simulate AD Samples
       *********************************************************************/
      costate
      {
          for(channel = 0; channel <= 7; channel++)
             ad_inputs[channel] = (rand()-0.5)/10 + 5567;

         waitfor( DelayMs(500) );
      }
[...]
```

Y esto es todo lo digno de ser nombrado, para este archivo.

El módulo iweb.lib

Acá tenemos algunos puntos para comentar que deberían terminar de redondear algunas ideas.

Las variables que queremos exponer al mundo vía web, deben ser accesibles al software RabbitWeb que es parte de nuestro servidor HTTP, hacemos esto con la directiva especial `#web`

```
[...]
#web ad_inputs
#web NAME
[...]
```

Además tenemos algunas variables para el uso propio de las funcionalides web del programa: `timeRefresh` que funcionará como una variable de control y `strDateTime` que contendrá el string de la fecha/hora actualizado. (Recuerden `<datetime> <?z print(@strDateTime) ?> </datetime>`)

```
[...]
char timeRefresh;
#web timeRefresh
char strDateTime[20];
#web strDateTime
```

Algo habíamos visto sobre la carga de los archivos en flash y la obtención del puntero a su posición de memoria:

```
[...]
#ximport "IWeb\index.zhtml"          index_html
#ximport "IWeb\values.xml"           values_xml
[...]
```

Y lo que sigue merece una explicación más detallada. Acá tenemos dos funciones `time_refresh` y `sync_time` y más abajo, en verde, dos directivas `#web_update`, que vinculan sendas variables (`timeRefresh` y `strDateTime`) con dichas funciones.
Lo que sucede entonces es que el software de RabbitWeb está al tanto de que, cuando un *HTTP request* modifica el valor de esas variables debe llamar posteriormente a las funciones indicadas. Entonces, ahora recuerden el momento de armar el *HTTP request* mediante el método POST, y es aquí cuando se vinculan ambos conceptos y operaciones. Si volvemos a la función de AJAX:

```
var params = (syncTime)? "strDateTime=" + getTime(): "timeRefresh=1";
[...]
xmlHttp.send(params);
[...]
```

Optábamos por dos maneras de armar el parámetro, elijiendo enviar `"strDateTime=" + getTime()"` o `"timeRefresh=1"`. Con lo cual, forzamos a la ejecución de una u otra función, en el primer caso realizamos la puesta en hora del RTC, en el segundo caso obtenemos la fecha y hora actualizadas para mostrar en la página. Estos procedimientos no serían necesarios si solo requiriésemos mostrar variables vía web, en este caso estamos realizando procesos adicionales antes y/o despues de las transferencias de datos y es por eso que necesitamos esta forma de señalización que permita saber cuando realizar determinada tarea.

```
/***************************
  clock control
***************************/
void time_refresh(void)
{
   get_time( strDateTime, &strDateTime[11]);
   strDateTime[10]=' ';
   timeRefresh=0;
}

void sync_time(void)
{
       set_time( strDateTime );
}
/*****************************************
 linking variables with each handler function
*****************************************/
#web_update timeRefresh              time_refresh
#web_update strDateTime              sync_time
```

La vinculación de la extensión del archivo que el browser solicita con su tipo MIME y con el handler que va a manejar el despacho del recurso:

```
[...]
SSPEC_MIMETABLE_START
   SSPEC_MIME_FUNC(".zhtml", "text/html", zhtml_handler),
   SSPEC_MIME_FUNC(".xml",   "text/xml",  zhtml_handler),
[...]
```

Y finalmente la vinculación de cada recurso con su posición en memoria, utilizando el puntero obtenido luego de usar `#ximport` :

```
[...]
SSPEC_RESOURCETABLE_START
[...]
   SSPEC_RESOURCE_XMEMFILE("/index.zhtml", index_html),
   SSPEC_RESOURCE_XMEMFILE("/values.xml", values_xml),
[...]
```

Y ahora sí, esto es todo lo que por el momento tenemos para decir sobre la forma de utilizar módulos Rabbit sumándoles funcionalidad web mediante su servidor HTTP, aprovechando su lenguaje de script del lado servidor e incorporando las ventajas de la metodología planteada por AJAX.