OCCAM'S RAZOR

ISSN: 1998-0537

Porque lo más sencillo es lo más probable

Número 5 · Invierno 2010

HARDWARE ABIERTO
La Nueva revolución

librerías
Visualización de Modelos 3D con OpenScenegraph

mú rápido
Analizamos el gusano de Morris

reverso
El Increíble Programa Menguante

Desafío Criptográfico
Desvelamos "la Clave" de Nuestro Desafío

NO TE CORTES CON LA NAVAJA DE OCCAM

SUMARIO

4 **DESAFÍO CRIPTOGRÁFICO**
Desvelamos "la Clave" de Nuestro Desafío

7 **LA COMUNIDAD DE OCCAM**
lista de Corre, Vídeos y Más
Os esperamos en nuestra lista de correo y canal de youtube

8 **RATAS DE BIBLIOTECA**
Visualización Fácil de Modelos 3D
Sólo tres líneas con OpenScenegraph

10 **MÚ RÁPIDO**
El Programa Fantasma
Analizamos el Gusano de Morris

16 **EN LA PRÁCTICA**
Hardware Abierto
La Nueva Revolución

23 **REVERSO TENEBROSO**
El Increíble Programa Menguante
Como Hacer Programas Muy Pequeñitos

28 **LA CACHARRERÍA**
Introdúcete en La Realidad Virtual
Primeros Pasos Para un DataGlove Casero

31 **TRUCOS**

Esta revista ha sido realizada con:

blender · GIMP · Emacs · Vim · LaTeX

Occam's Razor

Número 5. Invierno 2010

Dirección:
David Martínez Oliveira

Editores:
David Martínez Oliveira
Fernando Martín Rodríguez

Colaboradores:
Fernando Martín Rodríguez, Laura I. Rodríguez González, Er Tresdé, Casper van der Guorm, Er Che vi Gan, Er Mangante Mengunate, er Cortaó de Yervas

Maquetación y Grafismo
Laura I. Rodríguez González

Publicidad
Occam's Razor Direct
occams-razor@uvigo.es

Impresión
Por ahora tu mismo... Si te apetece

Consulta la página 32 para las excepciones a esta licencia

Editorial

Feliz 2011!

by The Occam's Razor Team

No podía terminar 2010 sin que Occam's Razor diera señales de vida. Así que aquí estamos, en el límite con una pequeña revista que esperamos os entretenga durante el pequeño "break" navideño, e intentando mantener nuestra periodicidad de año y medio :).

Aunque la mayoría de vosotros estaréis leyendo estas líneas en 2011, lo cierto es que este número 5 se ha terminado el 31 de Diciembre de 2010. Hemos hecho un pequeño esfuerzo para que nuestra periodicidad, "en la práctica", de año y medio no se alargue más.
Este número es un poco más corto de lo habitual, pero esperamos que os guste tanto como los anteriores. En él encontraréis, un par de artículos sobre software y un interesante dossier sobre el "nuevo" movimiento de hardware libre.
Las ratas de biblioteca nos introducen esta vez en el mundo de los gráficos 3D y la realidad virtual, junto con los chicos de la cacharrería que nos cuentan como empezar a construir un *Dataglove*.
En este número hemos sustituido "El Rincón de los Lectores" por una nueva sección que hemos llamado "Comunidad". En esta nueva sección no solo nos haremos eco de vuestros mensajes, sino también de los eventos más interesantes en nuestra lista de correo y en nuestro canal de YouTube. Esperamos veros a todos por allí.

Deseamos que la distrutéis y ....

Feliz 2011!

The Occam's Razor Team

La solución

Desvelamos "la clave" de nuestro desafío

por Fernando Martín Rodríguez

Estaba claro que tras el desafío del número anterior teníamos que publicar la solución. Allá va.

EL PROGRAMA DESCIFRADOR

Los profesores sabemos que en cualquier ejercicio puede haber diferencias grandes de dificultad según la forma de redactar el enunciado y también según qué apartados se pidan y en qué orden. En el caso que nos ocupa queremos descifrar un criptograma. Lo más fácil es empezar por analizar el "cifrador" y crear un descifrador.

El cifrador eran dos funciones de matlab... repito aquí la principal:

```
function criptograma = CifrarTexto(llano,clave)
raiz = floor(sqrt(clave));
resto = clave-raiz^2;
N = length(llano);
for i=1:N,
    [raiz, resto, digito] =
       DigitoDecimalRaizCuadrada(raiz,resto);
    cfr = llano(i)+digito;
    if cfr>90
        cfr = (cfr-91)+65;
    end
    criptograma(i) = char(cfr);
end
```

La línea 8 ($llano(i) + digito$) hace un trabajo fundamental. El programa obtiene una serie de dígitos (números entre 0 y 9) y los suma al texto en llano. Estamos suponiendo que el texto en llano son caracteres es ASCII y que sólo usamos mayúsculas, esto es: llano(i) va de 65 a 90 (la condicional que sigue asegura que la suma sea circular). Visto esto parece claro que si restamos en vez de sumar deberíamos tener un sistema descifrador. Esto es:

```
function llano = DesCifrarTexto(criptograma,clave)
raiz = floor(sqrt(clave));
resto = clave-raiz^2;
N = length(criptograma);
for i=1:N,
    [raiz, resto, digito] =
        DigitoDecimalRaizCuadrada(raiz,resto);
    cllano = criptograma(i)-digito;
    if cllano<65
        cllano = (cllano-65)+91;
    end
    llano(i) = char(cllano);
end
```

Lógicamente debemos volver a tener en cuenta la circularidad.

EL FUNDAMENTO DEL MÉTODO

Supongo que estaréis esperando a que explique qué hace la función auxiliar que genera los dígitos (aunque su nombre es bastante transparente). Recordemos el fuente:

```
function [raiz, resto, digito] =
            DigitoDecimalRaizCuadrada(raiz0,resto0)
%Obtener el siguiente decimal de una raiz cuadrada
resto1 = resto0*10;
resto2 = resto1*10;
aux1 = 2*raiz0;
digito0 = floor(resto1/aux1);
for cont = digito0:-1:0,
    aux2 = aux1*10+cont;
    aux3 = aux2*cont;
    resto = resto2-aux3;
    if (resto>=0)
        digito = cont;
        break;
    end
end
raiz = raiz0*10+digito;
```

El método funcionará si somos capaces de generar números (dígitos) "aparentemente" aleatorios a partir de la clave. Esto es: conociendo la clave debe ser fácil generar la secuencia mientras que en caso contrario debe parecer aleatoria. Como podéis deducir del nombre de la función a mí se me ocurrió utilizar la "aleatoriedad" de los decimales de un número irracional. De clave, por tanto, nos vale un número entero que no sea cuadrado perfecto.

Hemos implementado el algoritmo de la ráiz utilizando un par de funciones Matlab

Fijaos que quiero los dígitos exactos, sin redondeos, para eso utilicé el "muy olvidado" algoritmo de la raíz cuadrada (que todos estudiamos en el colegio alguna vez). Como todo, se puede encontrar en Internet:

Algoritmo de la raíz

Realmente esta forma de hacerlo puede tener sus "cosquillas"... fijaos que el método dejará de ser exacto al mismo tiempo que la aritmética entera de matlab deje de serlo. En el programa, los números decimales siempre se multiplican por una potencia de 10 para que sean enteros pero está claro que cualquier sistema perderá precisión al trabajar con muchos dígitos, lo que va a ocurrir con mensajes largos.

LA SOLUCIÓN

Pues a partir del criptograma:

DWSGNGGJBVVPIWYXXKJSGHVLLPX
VHQYWYAJBTLEDNLAROLXTBGINPF
ZYJRPXUMUFCVCJTMXRDKXIFUXRW

La idea era probar por fuerza bruta hasta lograr un texto llano coherente. Para no liarlo mucho decidí elegir una clave menor que 100 con lo que no había mucho que probar. Además, algo dentro me decía que el sistema podría ser más seguro si la clave es un número primo (je, je, esto no hacía falta adivinarlo).

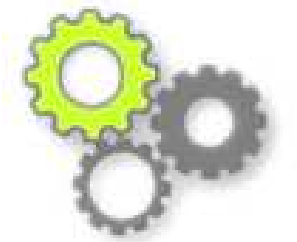
Probando con clave = 67 obtenemos un llano no sólo coherente sino conocido: "CONDIEZCANONESPORBANDAVIENTOENPOPAATODAVELANOCORTAELMARSINOVUELAUNVELEROBERGANTIN" (ver "Canción del Pirata", poema de José de Espronceda).

Sí, sí, se eliminaron los espacios, los acentos y las eñes. No debería ser muy difícil tener eso en cuenta... O complicar el método para tener desplazamientos mayores que 9 (agrupando dos dígitos, por ejemplo).

LOS ACERTANTES Habíamos prometido publicar sus nombres, allá van:

- Bultza (pseudónimo).
- Jorge Muñoz Castañer.
- Sergio Mendieta.

Enhorabuena! ■



Las Soluciones

Como han resuelto el problema los acertantes

Desafío Criptográfico I

enviado por Jorge Muñoz Castañer

27 de Agosto de 2009

Hola Fernando,

Pues nada... que me he puesto con el enigma criptográfico y lo he sacado. Te cuento:

Nombre completo: Jorge Muñoz Castañer

Criptograma descifrado:

CONDIEZCANONESPORBANDAVIENTOENPOPA
ATODAVELANOCORTAELMARSINOVUELAUN
VELEROBERGANTIN

Valor de la clave: 67

Una explicación breve de lo que hace el método y de cómo lo habéis atacado:

El método coge cada carácter, su código ASCII, y le suma un dígito de la raíz cuadrada de la clave. Al primer carácter el primer dígito y así en adelante. Si es necesario se calculan decimales de la raíz para tener dígitos suficientes.

Para atacarlo he usado el poco ortodoxo método de "fuerza bruta con diccionario inteligente". Primero escribí la función que desencripta un mensaje sabiendo la clave y la dejé corriendo con enteros sucesivos como clave. Para sacar el texto en claro inicial me centré en la primera palabra del mensaje. Por cómo está escrito el cifrador sólo se pueden usar en el texto letras mayúsculas y cada letra del mensaje cifrado solamente se puede corresponder con ella misma o una de las nueve anteriores (porque se le suma una cifra). Así que la primera palabra tenía que comenzar con U, V, W, X, Y, Z, A, B, C o D. Probé con algunas palabras (usando grep "a pelo" sobre el fichero de posibles mensajes) hasta que di con un texto con sentido. Si llego a ver que tardaba mucho así me habría puesto a ver los caracteres posibles para el segundo carácter del mensaje y así sucesivamente.

Aprovecho para felicitar a los redactores, editores y colaboradores de esta revistaza (vosotros sí que sois unas "malas bestias" ;-).

Un abrazo,

Jorge Muñoz

PD: Aprovecho y os comento algunos articulillos que si tenéis a bien podíais trabajar para el próximo número:

Sistemas empotrados Open Source y Hardware (Arduino, por ejemplo)

Intro a las FPGAs y OpenCores ■

Desafío Criptográfico II

enviado por Bultza

18 de Septiembre 2009

Mi enhorabuena por la revista, vi el anuncio en barrapunto y me ha encantado. aquí la solución:

Mensaje llano:

CONDIEZCANONESPORBANDAVIENTOENPOPA
ATODAVELANOCORTAELMARSINOVUELAUN
VELEROBERGANTIN

Clave: 67

El método que usais es sencillo pero interesante, se trata de una sustitución cesar utilizando en vez de un número constante de dígitos, los decimales de una operación de hacer la raiz cuadrada sobre la clave (es decir los decimales de un número irracional). El problema de este método es que como el usuario meta un número que sea cuadrado perfecto entonces el texto encriptado es igual al mensaje llano.

Estaba estudiando el método para romperlo y se me había ocurrido atacarlo a mano, es decir, es una sustitución cesar muy sencilla y una letra encriptada estará no más lejos de 10 dígitos, con eso y en algún punto del mensaje se podría encontrar alguna palabra, una vez hecho esto buscar un número de forma bruta cuyos decimales coincidiesen con los números encontrados y así saber el resto de desplazamientos y la clave. Era un método horriblemente inatractivo. Las primeras suposiciones tras probar yo mismo el método eran que el mensaje vuestro era en mayúsculas y sin espacios o tendríamos carácteres raros por todo el criptograma

Otra idea es que algunas letras no iban a ser encriptadas, cada letra tiene una probabilidad del 10 % de no ser encriptada, en un mensaje largo se podría encontrar incluso palabras completas.

Sin embargo he de reconocer que he descubierto el mensaje y la clave de forma bruta con el siguiente código:

```
%Descifrar criptograma Fuerza Bruta:
function DescifrarTextoBruto (criptograma,
                              deDonde,aDonde)
  for x = deDonde:1:aDonde
    clave = x;
    raiz = floor(sqrt( clave ));
    resto = clave - raiz^2;
    N = length(criptograma);
    for i=1: N,
      [ raiz , resto , digito ] =
        DigitoDecimalRaizCuadrada(raiz,resto);

      cfr = criptograma(i) - digito;
      %disp(cfr);
      if cfr < 65
        cfr = ( cfr + 91) - 65;
      end
     llano(i) = char(cfr);
    end
 disp(clave);
 disp(['llano: ' llano]);

end
```

Que lo único que hace es probar claves e imprimirlas cómodamente en la pantalla. Lo hice para descifrar mis mensajes y ver que funcionaban, había intentado descrifrar las primeras 10 letras, para ello he querido primero comprobar que vuestro mensaje si lo descrifraba con cualquier otra clave siempre me iba a dar para la letra D, siempre letras desde T hasta Z y desde A hasta D, así que imprimí las 100 primeras y me golpeó al ojo el CONDIEZ pues pensaba que estaba leyendo mi propio apellido, y coño, me salió un "mierda!.^{en} vez de un "toma ya", pues no era mi intención descubrirlo de esta forma, de hecho pensaba que habríais utilizado un número con decimales pero tampoco sería cuestión de hacer eso...

Un método más lógico hubiese sido empalmar mi programa de arriba con una comprobación con un diccionario de palabras que siempre aparecen tipo çonde.^{et}c pero eso sí que podría ser un proceso muy largo.

Voy a seguir pensando algún método más matemático porque seguro que lo hay, tengo ganas de leer una respuesta convincente a esto pero mi cabeza no da para mucho más así que no esperéis que llegue a ningun sitio ;).

Un saludo ■

Desafío Criptográfico III

enviado por Sergio Mendieta

Septiembre 2009

Buenos Días Señor Fernando Martín, Disculpe la interrupción, es para hacerle una pregunta sobre el tema "Un Desafío Criptográfico" de la revista "OCCAM's Razor",

La verdad, tal vez solo este algo perdido, veamos:

La raiz(67) = 8.**185352771872449969953703724 7339**29458880486815498039963066 7

Resalto hasta el carácter 31 después del punto dado que es hasta allí lo que nos da la calculadora de window, pero como indica usted en el desafió utiliza matlab, por lo tanto tal vez esté manejando mas dígitos después del carácter 31, que es lo que indica las funciones de cifrado, con lo cual cifraría todo el mensaje o sea los 81 caracteres.

Pero el resultado aplicando el descifrador:

clave=67(C),

criptograma=DWSGNGGJBVVPIWYXXKJS
GHVLLPXVHQYWYAJBTLEDNLAROLXTBGIN
PFZYJRPXUMUFCVCJTMXRDKXIFUXRW

llano=**CONDIEZCANONESPORBANDA VIENTOENP**UPWE...

Lo resaltado parece coherente (tal vez sea un mensaje para despistar), pero después de esto vienen un conjunto de letras que no lo son, ahora la función de cifrado no indica que se realicen otras operaciones, como el cambio de alfabeto, o el uso recursivo de la misma, etc

Mi pregunta es si el criptograma está correcto? lo que me ayudaría a seguir, no me de mas pista, solo afirme que el criptograma este bien.

En todo caso si el criptograma esta bien, el cifrado se debió realizar de esta forma cifrarTexto(texto1+cifrarTexto(texto2, clave2), clave1), pero esto son solo ideas.

Le agradezco su atención, creo que este correo va a ser inentendible! :(■

–

Si bien Sergio no resolvió completamente el desafío debido a errores de redondeo en las herramientas que utilizó, hemos considerado como válida su respuesta.

La Comunidad de Occam

Lista de Correo, Vídeos y más.

por The Occam's Razor Team

NUESTRA LISTA DE CORREO

Todavía en sus primeros pasos, la lista de correo de Occam's Razor ya lleva funcionando más de un año con unos 65 miembros oficiales. La lista es totalmente abierta así que cualquiera puede leer las discusiones sin necesidad de suscribirse.

Aquí tenéis alguno de los temas que hemos tratado en la lista.

Hacks de media tarde. WebCam 3D

Un sencillo programa para conseguir imágenes anaglifas utilizando dispositivos *v4lloopback*. Con un pequeño programa leemos fácilmente imágenes de dos cámaras, las procesamos y las hacemos disponibles a otras aplicaciones como si se tratase de una tercera cámara.

Más...

Lua en 5 minutos

Vimos como añadir un interprete LUA a vuestras aplicaciones en 5 minutos. Con unas pocas líneas de código añadimos a nuestros programas la capacidad de ejecutar scripts LUA y de hacer que esos scripts ejecuten funciones definidas en nuestros programas.

Mas...

Compressed sensing

Hablamos sobre compressed sensing y superresolución y proporcionamos algunos enlaces para introducirnos en el tema.

Mas..

Mensajería Instantánea Mejorada

Una versión mejorada del sistema de mensajería instantánea que introdujimos en el artículo de Netcat en el número 1 de Occam's, utilizando OSD (On-Screen Display) y un sencillo interfaz gráfico con *Xdialog*.

Mas..

Time-Lapse Videos con mplayer y ffmpeg

Un sencillo script que usa `mplayer` para capturar imágenes con una webcam cada cierto tiempo, para finalmente generar un video a partir de todas esas imágenes utilizando `ffmpeg`. Sabíais como utilizar el modo remoto de `mplayer`?

Mas..

Los miembros de la lista reciben la revista antes que nadie y además tienen la posibilidad de ofrecer sus comentarios antes de que se publique oficialmente. Además, estamos preparando nuevas actividades relacionadas con la revista.

NUESTROS VIDEOS

En nuestro canal de YouTube hemos publicado dos vídeos. El primero es una modificación del artículo de video vigilancia del número 4, en la que las cámaras se configuran para obtener una visión panorámica.

El segundo vídeo que hemos añadido fue generado con el script aparecido en la lista de correo para generar "time-lapsed" videos.

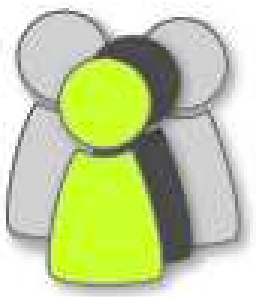
Si habéis producido algún video a partir de los artículos de Occam's Razor nos encantaría conocerlos y añadirlos como favoritos en nuestro canal o como video respuestas si están relacionados con alguno de los videos ya disponibles.

UNIROS A LA COMUNIDAD!

Nos vemos en nuestra lista de correo y canal de youtube.

Grupo Occam's Razor
Canal YouTube

OS ESPERAMOS!!!!



Visualización Fácil de Modelos 3D

Sólo 3 líneas con OpenScenegraph

por Er Tresdé

Cargar y visualizar modelos 3D *a mano* no es algo que podamos contar en una sola página de Occam's Razor. Sin embargo, con la ayuda de una librería (o conjunto de librerías para ser más exactos) como OpenScenegraph, se puede convertir en una tarea trivial. En este artículo os contaremos como visualizar modelos 3D utilizando esta librería en solo 3 líneas de código!

OpenScenegraph es una de las muchas librerías disponibles para el desarrollo de aplicaciones 3D. Esta en concreto se suele considerar la sucesora de la increíble SGI Performer, desarrollada por Silicon Graphics hace algunos años y que ya ha dejado de existir. Hay otras muchas, más orientadas al desarrollo de juegos (Ogre3D, Irrlicht, ...) o con modelos de programación más alejados de Performer.

En cualquier caso, OpenScenegraph ofrece muchísimas funcionalidades y volveremos a hablar de ella, en más detalle, en próximos números. Por ahora, veamos como mostrar un modelo 3D.

OpenScenegraph es un conjunto de librerías muy potente para el desarrollo de aplicaciones gráficas 3D

EL CÓDIGO

Aquí están las prometidas tres líneas de código (claro 3D... 3Líneas :).

```
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>

using namespace osgDB;
int main(int ac, char **a)
{
  osgViewer::Viewer viewer;
  viewer.setSceneData(readNodeFile(a[1]));

  return viewer.run();
}
```

Más fácil no se puede. Nuestro programa tiene una única variable, el visor (`viewer`). Esta clase se encarga, si nosotros queremos, de todo; crear la ventana, calcular la posición de la cámara, manejar los eventos de ratón y teclado,...

Tres líneas son suficientes para visualizar modelos 3D

Lo único que necesitamos decirle es que datos queremos visualizar y ponerla en marcha (con el método `run`).

Este sería el Makefile para compilar nuestro sencillo visor.

```
MY_CFLAGS=-I${DEV_DIR}/include
MY_OSG_LIBS=-L${DEV_DIR}/lib -losgViewer

mini: mini-viewer.cpp
	g++ -o $@ $< ${MY_CFLAGS} ${MY_OSG_LIBS}
```

En este *makefile* estamos usando una variable de entorno de soporte. Si habéis instalado OSG en el directorio por defecto, simplemente dadle el valor `/usr/local/`. Sino, probablemente sabréis lo que estáis haciendo y no necesitáis más ayuda :P.

UN POQUILLO DE SALSILLA

Aunque el programa mola (3 líneas y podéis ver un montón de modelos 3D), es un poco soso. Así que vamos a modificarlo para hacer que nuestro modelo rote.

Básicamente lo que haremos será crear un grafo de escena (efectivamente, un "scene graph") y sustituir la llamada a `viewer.run()` por nuestro propio bucle, de forma que podamos hacer rotar nuestro modelo 3D antes de pintar cada cuadro.

Para ello tenemos que añadir a nuestro grafo un nodo que nos permita transformar nuestro modelo, girarlo en este caso. De las muchas posibilidades que nos ofrece OSG, `PositionAttitudeTransform` es la más sencilla de utilizar.

Una vez creado este nodo, podemos añadir tantos nodos hijo como queramos, y todos ellos se verán afectados por nuestra transformación.

Con este nuevo grafo, ya solo tenemos que actualizar la orientación del nodo de transformación en cada iteración de nuestro bucle principal, antes de pedir a OpenScenegraph que pinte nuestra escena.

```
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <osg/PositionAttitudeTransform>
#include <osgGA/TrackballManipulator>

using namespace osg;
using namespace osgGA;
using namespace osgViewer;

int main(int argc, char **argv)
{
    ref_ptr<Node> model;
    Viewer        viewer;
    PositionAttitudeTransform *pat;
    TrackballManipulator *manipulator;
    float rot = 0.0f;

    manipulator = new TrackballManipulator();
    viewer.setCameraManipulator(manipulator);

    model = osgDB::readNodeFile(argv[1]);
    pat = new PositionAttitudeTransform;
    pat->addChild (model);

    viewer.setSceneData (pat);

    while (!viewer.done())
      {
        rot += 2.0;
        pat->setAttitude (Quat(DegreesToRadians(rot), Vec3(1,0,0) ,
                               DegreesToRadians(0.0), Vec3(0,1,0),
                               DegreesToRadians(rot), Vec3(0,0,1)));

        viewer.frame ();
        usleep (5000);
      }
}
```

POSICIÓN Y ORIENTACIÓN

Los nodos `PositionAttitudeTransform` son los más sencillos de utilizar. Una vez creados, podremos añadir nodos hijos con el método `addChild`. De hecho, este método lo proporciona la super clase |verb!Group!, pero no vamos a profundizar en el modelo OO de la librería.

Los dos métodos más importantes que nos proporciona esta clase son:

- `setAttitude`, que nos permite definir la orientación (rotación) para la transformación que estamos definiendo.
- `setPosition`, que nos permite definir la traslación para la transformación que estamos definiendo.

El método `setAttitude` espera como parámetro un quaternion. En nuestro ejemplo, lo creamos dinámicamente a partir de ángulos de Euler. Como podéis ver en el código, al crear nuestro quaternion, le pasamos como parámetros el ángulo que queremos rotar en cada uno de los ejes estándar. De la misma forma el método `setPosition` espera un objeto del tipo `Vec3`.

Ahora ya solo necesitamos crear un bucle, y actualizar la orientación de nuestra transformación en cada cuadro, y luego decirle a OSG que pinte ese cuadro llamando al método `frame` de la clase `Viewer`.

MANIPULADORES DE CÁMARA

Hay dos líneas de las que casi no hemos hablado. Cuando utilizamos el método `run` del visualizador, la librería se encarga de que podamos mover la cámara y además, de situarla en una posición adecuada para que podamos ver la escena.

Al modificar nuestro programa para controlar como se ejecuta el bucle principal de la aplicación, hemos perdido esa funcionalidad. La forma de "reactivarla" (por decirlo de alguna forma), es añadiendo lo que se conoce como un manipulador de cámara.

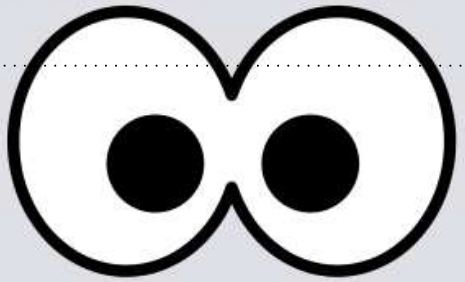
En nuestro ejemplo hemos utilizado un manipulador de tipo *Trackball*, el mismo que nos proporciona el método `run`.

Resumiendo, las dos líneas al principio de nuestro programa, crean ese manipulador y le dicen al visualizador (`viewer`) que lo use para controlar la cámara.

Si eliminái esas dos líneas, veréis que la cámara estará en la posición (0.0, 0.0, 0.0), al igual que el modelo, y por lo tanto, dependiendo del modelo 3D que estéis utilizando, puede ocurrir que no veáis nada. En ese caso, podéis utilizar el método `setPosition` de nuestro nodo de transformación para mover la cámara (o el modelo según se mire).

HASTA LA PRÓXIMA

Bueno, esperamos que esto os de para introduciros en el maravilloso mundo de los gráficos 3D. OpenScenegraph es una librería enorme, y ofrece una cantidad de herramientas monstruosa... horas y horas de diversión asegurada :). Hasta la próxima. ■



El Programa Fantasma

Analizamos el gusano de Morris

por Casper van der Guorm

Hace algunos años leí una noticia sobre el aniversario del gusano de Morris, ese que colapso una gran parte de internet allá por los 80. Quise saber un poco más sobre el tema y me encontré con algunas cosas curiosas que siguen funcionando en la actualidad. En este artículo os vamos a contar estas curiosidades creando.... EL PROGRAMA FANTASMA!!!

El programa fantasma es un sencillo programa que una vez ejecutado desaparece misteriosamente mientras intenta asustarnos con aterradores mensajes y terroríficos sonidos.

Nuestro programa fantasma desaparecerá misteriosamente una vez ejecutado

Para ellos vamos a utilizar algunas de las técnicas que usó el gusano de Morris hace algunos años. Para los que estéis interesados en el tema, comprobad la caja de recursos con algunos enlaces interesantes. No os vamos a hablar de los *exploits* que el gusano usó, ni de como se transfería de un ordenador a otros... simplemente vamos a hacer... un programa fantasma!

CAMUFLAGE

Bien, lo primero que vamos a hacer, es que nuestro programa fantasma sea invisible. Para ello vamos a utilizar distintas técnicas.

La primera consiste en cambiar el nombre de nuestro programa, de forma que la salida del comando `ps` no muestre nada sospechoso. A continuación, eliminaremos nuestras pistas del disco duro, y finalmente haremos que sea complicado encontrar nuestro PID.

Veamos como sería un programa como este para a continuación explicar en detalle cada una de estas características.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void limpia_disco (char *n) {
  printf ("Borrando Fichero\n");
}

static char *nombre_prg = "phantom";

int main (int argc, char *argv[]) {
  printf ("Hola soy el Programa Fantasma\n");
  limpia_disco (argv[0]);
  strcpy (argv[0], nombre_prg);

  while (1) {
    if (fork () > 0) exit (0);

    sleep (1);
  }

  return 0;
}
```

Efectivamente, la función que borra el ejecutable del disco no hace nada por el momento. Enseguida veremos como implementarla. Pero el resto de nuestras funcionalidades ya están ahí.

Como todos sabéis, el primer argumento que recibe cualquier programa es el nombre del programa ejecutado, así que si simplemente modificamos el valor de este parámetro, el nombre que, utilidades como `ps` , verá será precisamente el valor de ese parámetro. En nuestro caso estamos utilizando `phantom`, para poder identificarlo fácilmente. Para la versión final utilizaremos algo como `sh`. Bastante más discreto.

Luego entramos en el bucle principal de nuestro programa. Aquí haremos algunas cosas aterradoras, pero por el momento simplemente cambiamos nuestro identificador de proceso `PID`, cada segundo.

Modificando argv[0] podemos cambiar el nombre de nuestro programa

Para ello, solo tenemos que crear un nuevo proceso utilizando la llamada al sistema `fork`. `fork` creará una copia exacta del proceso que la ejecuta, así que si simplemente matamos el proceso original, podremos continuar nuestra ejecución con un PID diferente.

PROBANDO... PROBANDO

Vamos a probar el programa. Grabamos nuestro código en un fichero llamado `test.c` y compilamos con un sencillo `make test`.

Veamos que ocurre al ejecutarlo.

```
occams@razor$ ./test
Hola soy el Programa Fantasma
Borrando Fichero
occams@razor$ ps ax | grep test
31245 pts/7    R+     0:00 grep test
occams@razor$ ps ax | grep phantom
30620 pts/6    S      0:00 phantom
30623 pts/7    R+     0:00 grep phantom
occams@razor$ ps ax | grep phantom
30626 pts/6    S      0:00 phantom
30630 pts/7    R+     0:00 grep phantom
occams@razor$ killall test
```

Como podéis observar, hemos ejecutado nuestro programa de test, pero cuando listamos los procesos en ejecución filtramos la salida de `ps` con el nombre que habíamos elegido y no hay ni rastro del programa llamado `test`.

Además observad como en cada ejecución de `ps` el proceso `phantom` tiene un PID diferente. Bueno, por ahora vamos bien :).

OTROS USOS DE argv[0]

El uso más común de argv[0] lo encontramos en paquetes como *BusyBox* que ofrece un entorno shell bastante completo con un tamaño muy reducido. *BusyBox* genera un único ejecutable estático que incluye la funcionalidad de todos los comandos de este entorno (comandos como ls o chmod). Cuando el sistema se instala, se generan un montón de enlaces a este fichero, cada uno con el nombre de uno de los comandos implementados por el ejecutable principal.

La función principal del programa comprueba el valor de argv[0], el cual será distinto para cada uno de los enlaces creados y según su valor ejecutará una función u otra.

DESAPARECIENDO

Para que nuestro programa sea realmente un fantasma, tenemos que hacer que desaparezca del disco duro. Si hablamos de un gusano o virus real, borrar cualquier pista del disco duro hace más difícil el posterior análisis del código para la eliminación del *malware*.

Lo interesante de todo esto es que, una vez que nuestro programa está en ejecución, podemos borrar el ejecutable del disco y seguir funcionando. Es más, podemos leernos completamente en memoria y volver a escribirnos en el disco... lo que se llama la técnica Guadiana.

Veamos como de sencillo sería hacer esto.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <unistd.h>

static char *code = NULL;
static long  code_len = -1;
static char *nombre = NULL;

void limpia_disco (char *n) {
  struct stat buf;
  int         f;

  nombre = strdup (n);

  stat (n, &buf);
  code = (char*) malloc (
                 (code_len = buf.st_size));

  read ((f = open (n, O_RDONLY)),
        code, code_len);
  close (f);
  unlink (n);
}
```

El programa usa la llamada al sistema `stat` para averiguar el tamaño del ejecutable en el disco (`argv[0]`). Con esa información reserva un bloque de memoria donde almacenar el fichero y seguidamente lee el fichero y lo borra del disco. Esta última operación la hace el comando `unlink`.

El programa fantasma cambia continuamente su PID

En realidad no necesitaríamos leer el fichero en memoria, pero nosotros queremos hacer que el fichero reaparezca en el disco cuando el programa termine. De lo contrario lo perderíamos para siempre... bueno tendríamos que recompilarlo o recuperar una copia de seguridad.

APARECIENDO

Para hacer reaparecer nuestro fichero, simplemente tenemos que escribir los datos que acabamos de leer en el disco cuando el programa fantasma termine.

Eso lo haremos capturando la señal `SIGTERM` que el programa recibirá cuando alguien quiera que TERMine. Este sería nuestro manejador de señal.

```
#include <signal.h>

void kill_handler (int s)
{
  int f;

  write ((f = open ("/tmp/ghost-program",
                    O_CREAT | O_TRUNC | O_RDWR,
                    S_IRWXU )),
         code, code_len);
  close (f);

  exit (1);
}
```

Dos líneas de ná y estamos listos. Ahora solo necesitamos instalar el manejador. Eso lo hacemos en la función `main` en algún punto antes del bucle principal. Una línea como esta será suficiente.

```
signal (SIGTERM, kill_handler);
```

Y con estos últimos cambios, una ejecución del programa fantasma sería algo talque asín.

```
occams@razor$ ls -la
total 28
drwxr-xr-x  2 occams razor  4096 2010-01-03 12:43 .
drwxrwxrwt 14 occams razor 12288 2010-01-03 12:40 ..
-rwx------  1 occams razor 11249 2010-01-03 12:43 test01
occams@razor$ ./test01
Hola soy el Programa Fantasma
Borrando Fichero
occams@razor$ ls -la
total 16
drwxr-xr-x  2 occams razor  4096 2010-01-03 12:43 .
drwxrwxrwt 14 occams razor 12288 2010-01-03 12:40 ..
occams@razor$ ps ax | grep phatom
17148 pts/4    R+     0:00 grep phatom
occams@razor:/tmp/test$ ps ax | grep phatom
17154 pts/4    R+     0:00 grep phatom
occams@razor$ killall test01
occams@razor$ ls -la
total 28
drwxr-xr-x  2 occams razor  4096 2010-01-03 12:43 .
drwxrwxrwt 14 occams razor 12288 2010-01-03 12:40 ..
-rwx------  1 occams razor 11249 2010-01-03 12:43 test01
```

En esta secuencia de comandos, podemos ver como el programa desaparece el disco duro, cambia su nombre y pid continuamente y vuelve a aparecer misteriosamente cuando lo *matamos*.

Inquietante no?... Pero todavía nos falta un detalle para que el programa fantasma resulte realmente aterrador...

EL ARCHIVO AUTO-EXTRAÍBLE MÁS TONTO DEL MUNDO

El toque final que necesita nuestro programa es un sonido que meta miedo, un sonido terrorífico. Quizás podríamos generarlo sintéticamente, pero en lugar de eso simplemente vamos a reproducir algún fichero pregrabado de forma que tengamos máxima flexibilidad.

Por tanto, necesitamos añadir esos datos a nuestro programa fantasma. No quedaría muy bien un programa como este que hubiera que instalar con el típico `configure` y copiara ficheros en `/usr/local`. Necesitamos un *archivo auto-extraíble* (o Self-extracting File).

El programa fantasma intentará asustarnos con un terrorífico sonido.

Básicamente lo que queremos es un instalador que, en un solo fichero, contenga todos los ficheros de la instalación y que cuando se ejecute se *pseudo-instale* y ejecute todo sin mediar palabra. Para ello, lo primero que necesitamos es un mecanismo para añadir datos extra a nuestro ejecutable... lo que los ingleses llamaría *payload* y que podríamos traducir por la más larga expresión *carga útil*.

La forma más sencilla de añadir datos a un fichero es utilizando el comando `cat`. Así, si el fichero que queremos incluir se llama `sound.mp3`, el siguiente comando nos solucionará la papeleta.

```
occams@razor$ cat test02 sound.mp3 > fantasma
```

Más fácil no se puede.

Ahora sólo necesitamos saber donde termina nuestro programa y donde empieza el fichero. Esta información la podemos calcular a partir del tamaño del fichero, que ya conocemos, puesto que lo necesitamos para leer en memoria todo el fichero y, una de dos; el tamaño del ejecutable o el tamaño del fichero que hemos añadido.

DESEMPAQUETANDO

La forma más sencilla de desempaquetar nuestro código es usando el tamaño del fichero que queremos empaquetar. Enseguida veréis porqué. Lo primero que tenemos que hacer en un *Makefile* para que nos resulte más fácil todo el proceso. Sería algo talque asín.

```
PAYLOAD=sound.mp3
P_SIZE=`ls -l ${PAYLOAD} | awk '{print $$5}'`

all:test03

test03: test03.c
	gcc test03.c -o kk -DPAY_SIZE=${P_SIZE}
	strip -s kk
	cat kk ${PAYLOAD} > test03
	rm kk
	chmod 777 test03
```

No olvidéis los *TABs* en la última parte del fichero. Lo que estamos haciendo es obteniendo el tamaño de nuestro *payload* utilizando el comando `ls` y filtrando con `awk`. Ese valor nos lo guardamos en una variable que luego pasaremos al programa como un `#define PAY_SIZE` cuando lo compilemos, utilizando la opción de compilación `-D`.

Ahora sólo necesitamos añadir una función que use esa información para generar un fichero que podamos reproducir más tarde. Esta es la función que necesitamos.

```
#define TFILE "/tmp/.xwsfsa"
void graba_fichero (void){
  int f, offset;

  offset = code_len - PAY_SIZE;
  write ((f = open (TFILE, O_CREAT | O_TRUNC |
                           O_RDWR, S_IRWXU)),
         code + offset, PAY_SIZE);
  close (f);
}
```

Sencillo no?. Simplemente calculamos el desplazamiento en el fichero donde empiezan los datos extra, y creamos un fichero en `/tmp`, oculto y con un nombre que lo haga parecer algo temporal :P. Luego simplemente grabamos esos datos en el fichero.

Ya casi estamos listos.

PROBANDO... OTRA VÉZ

Para poder probar nuestro programa fantasma, necesitamos añadir una llamada a nuestra función `graba_fichero` desde el programa principal. La podéis añadir en cualquier punto entre la llamada a `limpia_disco` (que es la que lee los datos en memoria, que necesitamos para poder extraer nuestro *payload*) y el bucle principal.

Si queremos que el programa fantasma sea supermisterioso, entonces necesitamos modificar nuestro manejador de señales. Si recordáis, cuando el programa recibe la señal `SIGTERM` de terminación, el fichero ejecutable vuelve a aparecer misteriosamente. Por esta regla de tres, lo suyo sería que el *payload* desaparezca.

Seguro que no tenemos que deciros donde poner el `ulink` verdad?

Ahora solo tenéis que elegir el mp3 que más os guste, copiarlo en el directorio con el nombre `sound.mp3` y ejecutar `make`.

```
occams@razor$ make
gcc test3.c -o kk1 -DPAY_SIZE=‘ls -l sound.mp3|awk ’{print $5}’‘
strip -s kk1
cat kk1 sound.mp3 > test03
rm kk1
occams@razor$ ./test03
Hola soy el Programa Fantasma
Borrando Fichero
occams@razor$ ls -la /tmp/.x*
-rwx------ 1 occams razor 473128 2010-01-07 19:25 /tmp/.xwsfsa

occams@razor$ file /tmp/.xwsfsa
/tmp/.xwsfsa: Audio file with ID3 version 23.0 tag, MP3 encoding
occams@razor$ killall test01
occams@razor$ ls -la /tmp/.x*
ls: cannot access /tmp/.x*: No such file or directory
```

Éxito rotundo!... Nuestro *payload* se genera correctamente y desaparece misteriosamente cuando nuestro programa muere... sin dejar rastro.

UN POCO MÁS GEEK

No ha estado mal, pero el método es bastante simplón y tiene el inconveniente de que cada vez que queramos cambiar el *payload* tendremos que recompilar el programa. Así que, fundamentalmente, tenemos dos alternativas; o guardamos en el fichero ejecutable su tamaño o lo calculamos en tiempo de ejecución.

El payload de nuestro programa fantasma se graba en /tmp con un nombre poco sospechoso.

En cualquiera de los dos casos tendremos que hurgar en la especificación ELF. Un rápido vistazo nos proporciona una serie de alternativas para almacenar nuestro tamaño de ejecutable (y por tanto el *offset* a los datos en el fichero).

- Utilizar alguna de las secciones que los compiladores generan y que *por ahora* no se utilizan. ¿Todavía no habéis leído Occam's Razor 3?... ¿a qué estáis esperando?
- Utilizar alguno de los huecos entre segmentos utilizados para el alineamiento.
- **Utilizar la cabecera ELF**

Ya os imagináis que vamos a utilizar eh?. Bien, la razón es que las dos primeras opciones son, a efectos prácticos iguales a nuestra primera versión, ya que la posición en la que almacenaremos los datos variará de un programa a otro... sin embargo, la cabecera siempre está ahí.

Podemos utilizar las características del formato ELF para almacenar datos en el propio ejecutable.

Examinando de cerca la cabecera (y haciendo algunas pruebas) es fácil encontrar un par de sitios donde almacenar un entero de 16bits (el programa fantasma es muy pequeño... casi no existe :). La mejor alternativa es justo al principio de todo, donde se almacena el número mágico.

La especificación de ELF nos dice que este número se encuentra al principio del fichero y lo interesante de este campo es que los últimos 7 bytes de este campo están reservados para aplicaciones futuras... bueno, la nuestra es una aplicación del futuro :).

PARCHEANDO ELFOS

Bien, el programa para almacenar un valor en los bytes reservados del campo de identificación de un ejecutable en formato ELF, sería algo como esto:

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define OFFSET 9

int main (int argc, char *argv[]) {
  char *prg = argv[1];
  int  fd, l = atoi (argv[2]);

  printf ("Fichero '%s' Off: %d Valor: 0x%x\n",
          prg, OFFSET, l);
  fd  = open (prg, O_RDWR, 0);

  lseek (fd, OFFSET, SEEK_SET);
  write (fd, &l, sizeof(int));
  close (fd);
}
```

Sí, eso es todo. Abrimos el fichero, nos movemos 9 bytes desde el principio (ahí es donde empiezan los bytes reservados), y escribimos el entero que queramos.

Veamos que tal funciona.

```
occams@razor$ readelf -h test03
ELF Header:
 Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 Class:                             ELF32
 Data:                              2's complement,little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              EXEC (Executable file)
 Machine:                           Intel 80386
 Version:                           0x1
 Entry point address:               0x80486d0
 Start of program headers:          52 (bytes into file)
 Start of section headers:          4444 (bytes into file)
 Flags:                             0x0
 Size of this header:               52 (bytes)
 Size of program headers:           32 (bytes)
 Number of program headers:         7
 Size of section headers:           40 (bytes)
 Number of section headers:         27
 Section header string table index: 26
occams@razor$ ls -l kk
-rwxr-xr-x 1 occams occams 5524 2010-01-07 19:23 kk1
occams@razor$ make patcher
gcc patcher.c -o patcher
occams@razor$ ./patcher test03  5524
Fichero 'test03' Off: 9 Valor: 0x1594
occams@razor$ readelf -h test03
ELF Header:
 Magic:   7f 45 4c 46 01 01 01 00 00 94 15 00 00 00 00 00
 Class:                             ELF32
 Data:                              2's complement,little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              EXEC (Executable file)
 Machine:                           Intel 80386
 Version:                           0x1
 Entry point address:               0x80486d0
 Start of program headers:          52 (bytes into file)
 Start of section headers:          4444 (bytes into file)
 Flags:                             0x0
 Size of this header:               52 (bytes)
 Size of program headers:           32 (bytes)
 Number of program headers:         7
 Size of section headers:           40 (bytes)
 Number of section headers:         27
 Section header string table index: 26
```

Ahora solo tenemos que ejecutar el programa para ver que no hemos roto nada. Bueno, esto es solo un comentario por si decidís jugar con otros campos de la cabecera...

La cabecera ELF es un buen lugar para almacenar algunos datos.

El último toque consiste en modificar nuestro programa y *Makefile* para utilizar esta información nueva información.

NUEVA VERSION

La nueva versión de `guarda_fichero`, es bastante sencilla, ya que hemos leído el fichero completo en memoria. Sólo tenemos que leer el entero almacenado en la posición 9. La función sería tan tonta como esto:

```
void graba_fichero (void){
  int f, offset;

  offset = *((int*)(code + 9));
  write ((f = open (T_FILE, O_CREAT | O_TRUNC |
                           O_RDWR, S_IRWXU )),
         code + offset, code_len - offset);
  close (f);
}
```

El último paso es modificar el *Makefile* para parchear nuestra aplicación.

```
PAYLOAD=sound.mp3
all: parcher temp04

patcher: patcher.c

test04: test04.c
        gcc test04.c -o kk
        strip -s kk
        ./patcher kk `ls -l kk | awk '{print $$5}'`
        cat kk ${PAYLOAD} > test02
        rm kk
        chmod 777 test02

}
```

Si todo ha ido bien, programa seguirá funcionando como hasta ahora, la diferencia es que ahora no tendremos que recompilar nuestro programa si modificamos nuestro fichero de *payload*. Es más, con nuestro nuevo esquema sería sencillo añadir más de un fichero con algunas modificaciones. ¿Alguien se anima?

EL FACTOR MULTIMEDIA

Ahora que ya hemos solucionado el problema de distribuir el fichero de sonido junto con el ejecutable, solo tenemos que añadir al programa fantasma la capacidad de reproducirlo. Ya os hemos contado en números anteriores de Occam's como reproducir ficheros MP3 de forma muy sencilla.

Sin embargo, en este caso, no nos vamos a complicar en absoluto y simplemente utilizaremos un reproductor multimedia estándar. En nuestro caso utilizaremos `mplayer`, pero `mpg321` o `vlc` son candidatos igual de buenos. Para ser rigurosos, el programa fantasma debería comprobar que reproductor está disponible en el sistema y auto-configurarse para utilizarlo. Eso os lo dejamos como entretenimiento para vosotros.

Con esta estrategia de simplicidad extrema, nuestro bucle principal pasaría a ser algo como esto.

```
int main (int argc, char *argv[]) {
  int audio = 0;

  limpia_disco (argv[0]);
  graba_fichero ();

  strcpy (argv[0], nombre_prg);
  signal (SIGTERM, kill_handler);

  while (1) {
    if (fork () > 0) exit (0);
    if (!sonido)
      {
        system ("mplayer -loop 0 "
                T_FILE " > /dev/null 2>&1 &");
        sonido = 1;
      }
    sleep (1);
  }

  return 0;
}
```

MAMA, MAMA, SIN MANOS

Hemos dejado esta parte para el final, en el más puro estilo "Zona Geek" de la revista. ¿Qué os parecería que el programa pudiera calcular la posición del fichero sin ayuda ninguna?

Pues claro. Mola!. Y además es muy sencillo.

Lo que sucede, es que la tabla de secciones ocupa la última parte del fichero ELF. Así que lo único que tenemos que hacer es calcular donde termina esa tabla, y sabremos donde termina nuestro fichero.

Todos los elementos para calcular este desplazamiento están en la cabecera. Básicamente necesitamos estos tres:

- `off`. Offset en el fichero al inicio de la tabla de secciones
- `shnum`. Número de secciones en el ejecutable
- `shsize`.Tamaño de una entrada en la tabla de secciones.

Así que el tamaño de nuestro fichero será:

$size = off + shnum * shsize$

Nuestra función graba fichero sería ahora algo así.

```
void graba_fichero (void) {
  int           f, offset;
  Elf32_Ehdr   *hdr = (Elf32_Ehdr*) code;

  offset = hdr->e_shoff +
           hdr->e_shnum * hdr->e_shentsize;
  write ((f = open (T_FILE, O_CREAT | O_TRUNC |
                            O_RDWR, S_IRWXU )),
         code + offset, code_len - offset);
  close (f);
}
```

Simplemente estamos utilizando un puntero a una estructura que representa la cabecera de un fichero ELF (`Elf32_ehdr`) para acceder a los datos que acabamos de leer del disco. Como lo que hemos leído es un fichero ELF, podremos acceder a su cabecera utilizando este puntero.

A continuación, simplemente calculamos la posición de nuestro fichero, como os acabamos de contar, y nos deshacemos de `ls` y `awk`.

Bueno, todavía hay una cosa que tenemos que contar.

El método que acabamos de describir funciona con ejecutables procesados por `strip`. La razón es que una de las cosas que hace `strip` es eliminar la tabla de símbolos del programa. Está tabla suele estar al final, después de las secciones, pero, desafortunadamente, no tiene una estructura tan uniforme, y el programa se haría un poco más complicado.

La verdad que esto último ni lo hemos probado, pero sería muy interesante que alguno de vosotros, si lo intenta, nos contara sus resultados.

OTRAS APLICACIONES

Asustar a la gente es una aplicación muy interesante, pero hay otras mucho más prácticas, como por ejemplo la instalación de programas.

Si en lugar de un fichero de sonido añadimos un `tar.gz`, y si en lugar de utilizar `mplayer`, utilizamos `tar`, obtenemos una forma muy sencilla de generar un instalador de aplicaciones.

Es cierto, que utilidades como `makeself.sh` ya hacen eso. ¿Pero y si queremos que nuestro instalador muestre unas bonitas ventanas, gráficos y música?

¿Se os ocurre alguna otra idea? ■

Hardware Abierto

La nueva revolución

por er Che vi Gan

En los últimos años una nueva perspectiva de la filosofía libre ha empezado a tomar cuerpo. Se trata del movimiento "Open Hardware" o "Hardware Abierto". En este artículo os contamos quienes son sus principales protagonistas.

Hoy por hoy todos estamos familiarizados con el software libre, o al menos con el software de fuentes abiertas, pero puede que el concepto de hardware abierto no nos resulte tan familiar.

En los últimos años un nuevo movimiento ha irrumpido en el escenario de los sistemas libres. Se conoce como "Open Hardware" y trata de aplicar las mismas ideas y filosofía del software libre en el mundo del hardware.

El movimiento Open Hardware trata de aplicar la filosofía del software libre al diseño del HW

Dicho de otra forma, los diseños electrónicos, mecánicos, las placas, de un determinado hardware se hacen disponibles libremente para nuestro disfrute, y nos proporcionan una forma más de aprender y aplicar nuestras propias ideas.

Este movimiento va unido a otro maravilloso cambio que hemos ido experimentando en estos últimos años; el acceso a un precio razonable a alta tecnología con la que experimentar en nuestras casas. En este artículo os presentaremos lo que, a nuestro humilde juicio, consideramos los proyectos más emblemáticos dentro de este nuevo movimiento.

ARDUINO

Arduino es quizás el mejor embajador del hardware abierto que podemos encontrar, y por ello es el primero del que vamos a hablar. Esta pequeña placa nos proporciona un fabuloso entorno de desarrollado además de los principales interfaces que vamos a necesitar en sencillas aplicaciones... en definitiva, para cosas normales hace nuestra vida muy fácil.

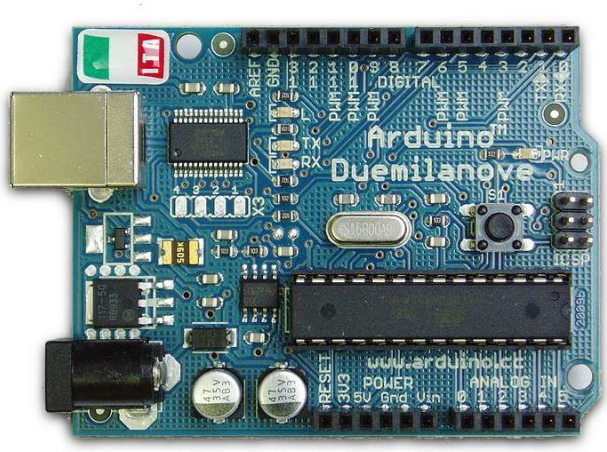
Existen multitud de versiones y variaciones de esta pequeña tarjeta, pero quizás la más conocida por todos es la denominada "Duemilanove".

En unas dimensiones de 6.8 x 5.3 centímetros, nos proporciona todo un procesador ATMega328, 32 KiB de memoria Flash, 1KiB de EEPROM, 2KiB de SRAM, 14 entradas y salidas digitales 6 de ellas permitiendo PWM y 6 entradas analógicas extra.

Todo un ejemplo de versatilidad por unos 28 EUR, y toda la información para poder construirla nosotros mismos.

Dos cosas tenemos que comentar sobre el Arduino.

La primera es la impresionante comunidad tras esta pequeña tarjeta, no solo el soporte ahí fuera, sino la cantidad de proyectos, información, y extensiones para nuestro Arduino. Estar tarjetas de extensión se conocen como "Shields" y las hay de todos los tipos: Ethernet, distintas opciones wireless, driver para motores, ...



La segunda cosa que hay que comentar sobre el Arduino es su increíble y fácil de usar entorno de desarrollo. El IDE desarrollado en Java es multiplataforma y el lenguaje (similar al C) es muy sencillo. En cuestión de minutos veremos a nuestro Arduino empezar a hacer cosas.

OPENMOKO

Si tuviéramos que decidir cual es el segundo proyecto Open Hardware, los teléfonos móviles de OpenMoko tienen que ser los elegidos. Si bien estos dispositivos no han gozado de la popularidad de muchos de los que vamos a nombrar en este artículo, lo que esta compañía consiguió fue bastante impresionante. La producción de un teléfono móvil totalmente abierto.

Esto supone un salto cualitativo con respecto a otros dispositivos como el Arduino, orientados a aficionados a la electrónica y el hacking. Un teléfono móvil comercial es una cosa seria.

Lamentablemente, y aunque el que escribe es un orgulloso propietario de un OpenMoko FreeRunner, la empresa tuvo que cerrar y, al menos, la parte comercial del proyecto se canceló.

OpenMoko, produjo dos tipos de terminales; el Neo 1973 y el Neo FreeRunner, siendo este último el modelo que se planteó como un producto comercial.

El Neo FreeRunner es un dispositivo increíble. Hace tres años te permitía arrancar Debian, con una pantalla de una calidad estupenda (todavía hoy me sorprende cuando lo enciendo), modem GSM, acelerómetros, GPS y wifi. Un completo ordenador con una potencia más que razonable en la palma de tu mano.

Desafortunadamente, algunos problemas técnicos y un mercado tan rápido como el de la telefonía móvil, dejó a OpenMoko fuera de combate y ahora solo la parte del proyecto relacionada con el software (OpenMoko Linux) existe.

El Openmoko Freerunner ha sido el primer smartphone libre de la historia

De cualquier forma, lo más importante es que estos señores demostraron que era posible hacerlo y el próximo que lo intente ya tendrá un buen camino andado.

Al igual que sucede con otros sistemas abiertos, cuanto más nos acercamos al hardware más difícil es hacer pública toda la información. En el caso de Openmoko, dos chips nos mantienen fuera de una plataforma totalmente abierta.

El primero de ellos es el chip encargado de los gráficos, conocido como "Glamo". Este chip supuestamente proporciona aceleración 3D sin embargo, para obtener esa información era necesario firmar un NDA con la compañía de productora del chip.

> Un NDA (Non-Disclosure Agreement o Acuerdo de no Revelado de información), permite a los desarrolladores obtener la información que necesitan para desarrollar su aplicación, pero no les permite desvelarla, por lo cual, en la mayoría de los casos, el software o hardware desarrollado con esa información no va a poder ser libre.

El otro componente sujeto a restricciones es el modem GSM. Si bien, el acceso al modem a través de su interface serie y comandos AT está documentado, los detalles internos del modem no. ¿Cuáles son esos detalles internos?, básicamente la cadena de radio del hardware, o dicho de otra forma, el control de lo que sale y entra por la antena al más bajo nivel.

De acuerdo a la información en la wiki de OpenMoko, proporcionar esa información permitiría hacer algunas perrerías a la red GSM que no estarían bien y que podría llegar a ser un grave problema.

Incluso con estas dos salvedades, los terminales de OpenMoko son lo más abierto que se ha creado nunca, y desde el punto de vista de un desarrollador proporcionan control absoluto sobre las funcionalidades que el teléfono ofrece.

BEAGLEBOARD

Si bien la BeagleBoard se puede decir que es una recién llegada al mundo del hardware abierto, tenemos que admitir que ha entrado por la puerta grande.

Por unos 125 USD más gastos de envío, obtenemos un potente ordenador en una placa de 7x7 cm. Además, por el mismo precio, adquirimos una completa plataforma de desarrollo embebido.

La BeagleBoard es una placa de bajo coste que incluye un OMAP3 y un conjunto mínimo de periféricos para poder trabajar con ella. Estos procesadores, o mejor dicho estos SOC (System On Chip) OMAP3 son muy interesantes por dos razones.

La primera es que es un chip bastante común en el mundo de la telefonía móvil, lo que hace de esta placa un sistema adecuado para el desarrollo de aplicaciones, o incluso prototipos basados en él.

La segunda es que, en un solo chip, tenemos un procesador ARM Cortex-A8, un DSP y un core gráfico con aceleración 3D (OpenGL ES 2.0), lo que nos permite explorar un sin fin de interesantes áreas.

La comunidad tras esta tarjeta es excepcional y el soporte que proporcionan es muy bueno.

Además del OMAP3 (un OMAP 3530), la tarjeta proporciona un puerto USB Host y otro USB OTG (con lo que podemos jugar a hacer nuestros dispositivos USB inteligentes), entradas y salida de audio, una salida de vídeo compuesto una salida DVI-D a través de un conector HDMI (para reducir espacio).

El OMAP3 incluye un procesador ARM, un DSP y un acelerador gráfico en un solo chip

Además de este hardware tan interesante, la tarjeta nos permite correr, casi cualquier versión de linux (gracias a varios proyectos de la comunidad) además de Android. Otros proyectos para soportar Windows Embedded o Symbian no están tan avanzados.

Si bien, el diseño de la tarjeta es totalmente abierto, desde este punto de vista, hay una pega relativa al sistema gráfico. El driver gráfico que controla el core SGX en el OMAP3 es libre, sin embargo, la librería OpenGL ES 2.0 en el espacio de usuario no lo es.

Con los últimos progresos con el DSP esas librerías son quizás el único escollo para disponer de un sistema hardware totalmente abierto.

Para terminar, la BeagleBoard incluye un puerto de expansión con el que poder acceder a varios de los interfaces del OMAP3 tales como I2C, MMC/SD o pines de entrada salida digital. Este conector ofrece un montón de posibilidades para conectar esta pequeña tarjeta con otros dispositivos, o crear los nuestros propios para ampliarla de acuerdo a nuestras necesidades.

SECUELAS DE LA BEABLEBOARD

Puesto que los desarrolladores de la BeagleBoard han proporcionado a la comunidad toda la información sobre como construir la tarjeta, hay unos cuantos clones, con sus pequeñas variaciones. En el cuadro de recursos al final del artículo podréis encontrar enlaces a varias de ellas.

Sin embargo, por secuelas, nos vamos a referir a dos proyectos concretos.

El primero de ellos es OpenPandora. Este proyecto consiste en el desarrollo de una consola portátil ejecutando GNU/Linux y haciendo uso de todos los emuladores de otras consolas disponibles para este sistema operativo.

Al igual que sucedió con OpenMoko, tener como objetivo el mercado de masas es un gran handicap para una pequeña empresa, sin embargo, tras muchos problemas y un increíble soporte por parte de la comunidad, las primeras OpenPandora están ya disponibles.

Desde un punto de vista técnico, una OpenPandora es una BeagleBoard en una caja estilo consola portátil, con teclado, pantalla táctil y wifi. A efectos prácticos un completo ordenador para llevar en el bolsillo, es decir, no necesitas usarla para jugar si no quieres :).

El otro proyecto que sí es una secuela de la BeagleBoard, es la BeagleBoard-XM, la segunda generación de la tarjeta.

Esta nueva tarjeta es un poco más grande y un poco más cara que la anterior. Bueno realmente tiene el mismo precio que tenía la BeagleBoard cuando salió al mercado hace un par de años, 149 dolares americanos.

Esta nueva versión utiliza una versión mejorada del SoC de la BeagleOriginal, un DM3730. Este chip ofrece lo mismo que el anterior OMAP 3530, pero mejorado. Un procesador ARM Cortex-A8 a 1GHz, un DSP a 800 MHz y una versión más potente del core gráfico SGX.

Además del nuevo cerebro de la tarjeta, los periféricos también se han mejorado. La BeagleBoard-XM incluye un hub USB en la propia placa, proporcionando 4 puertos USB, además incluye un interfaz de red RJ-45, el puerto serie ahora tiene un conector canon sub-D estándar, y se ha incluido un conector para acceder directamente al interfaz de cámara que proporciona el DM3730. El OMAP 3530 también disponía de este interfaz, pero en la BeagleBoard original no estaba disponible.

La BeagleBoard-XM además proporciona los mismos buses de expansión y de conexión de LCD que ya encontramos en las últimas versiones de la BeagleBoard original.

BEAGLEPHONES

También basados en la BeagleBoard, hay dos interesantes proyectos relacionados con la telefonía móvil.

El primero de ellos es el proyecto Wild Ducks, que busca desarrollar un diseño de referencia de un teléfono móvil ejecutando Symbiam3 a un precio asequible. El proyecto está progresando muy bien y en su página web podéis encontrar interesante información sobre como trabajar con Symbian o módulos GSM que podéis utilizar en otras aplicaciones.

Wild Ducks, OMBeagle y GTA04 son proyectos orientados al desarrollo de smartphones con diseños abiertos

Como valor añadido este proyecto se aprovecha de el sistema operativo de Nokia, Symbian, que ha sido liberado recientemente.

Otro proyecto con el que introducirnos en el mundo de la telefonía móvil es el OMBeagle o OpenMokoBeagle. Sí, este proyecto pretende crear un híbrido entre nuestro querido OpenMoko y la potente BeagleBoard.

Para ello han desarrollado una placa que actúa de interfaz entre la BeagleBoard y el OpenMoko, permitiendo utilizar la pantalla del OpenMoko como dispositivo de salida de la Beagle. En principio no tienen intención de utilizar el modem GSM en el teléfono, sino utilizar alguno de los modems UMTS USB disponibles en el mercado.

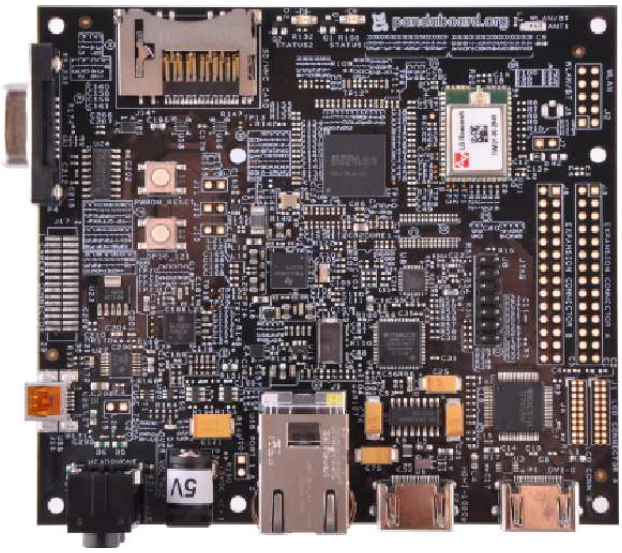
El mismo grupo detrás de OMBeagle ha iniciado el desarrollo del GTA04, la nueva generación de los terminales Openmoko. El GTA04 será un OpenMoko Freerunner en el que la placa principal se sustituirá por un nuevo diseño basado en la BeagleBoard... lo cual es muy interesante.

Hace escasa semanas, se acaba de iniciar un programa de "Adopción Temprana" (*Early Adopter Programme*) , que permite, a todos aquellos que lo deseen tener acceso a los primeros prototipos y por lo tanto comenzar el desarrollo y test de sus productos antes de que el nuevo terminal salga a la venta.

PANDABOARD

La PandaBoard es, por decirlo de alguna forma la hermana mayor de la BeagleBoard. Para hacernos una idea, es una mala bestia. Esta tarjeta está disponible desde verano del 2010 y promete.

Con la misma filosofía y por unos 174 USD, dipondremos de todo un OMAP4... si la nueva versión del OMAP3.



El nuevo OMAP4 nos ofrece dos cores ARM Cortex-A9, un nuevo sistema DSP también multicore y un sistema gráfico con un nuevo core SGX mucho más potente.

Además de todo eso, que nos lo da el OMAP4 de porque sí, la tarjeta nos proporciona dos puertos USB host y un puerto USB OTG, conector Ethernet y wifi-bluetooth en la misma placa. Además de todo eso, la tarjeta nos proporciona dos salidas de video, una HDMI y otra DVI-D en un conector HDMI (como en la BeagleBoard), además de un conector para una pantalla LCD similar al de la BeagleBoard y un interfaz para la conexión de una cámara como el que encontramos en la BeagleBoard-XM.

Esta es una tarjeta muy interesante ya que el OMAP4 probablemente sea uno de los procesadores que veremos en la próxima generación de móviles, compitiendo con los excelentes Snapdragon de Qualcomm.

Si la comparamos con la Beagle, no todo son ventajas. Bueno, eso dependerá de nuestra aplicación. Los principales factores a tener en cuenta son, por una parte, que esta tarjeta es un poco más grande (11x10 cm aprox) y por la otra que todo ese hardware extra en la tarjeta hace que consuma más potencia.

FOX BOARD

Una tarjeta mucho más veterana es la FOX board de la compañía ACME Systems. Con un precio similar y una tamaño de 6.6 x 7.2 cm, este pequeño PC nos ofrece un completo sistema GNU/Linux con dos puertos USB, un interfaz ethernet y puertos de expansión para conectarle lo que queramos.

Al igual que todas las demás, todos los esquemas electrónicos están disponibles en la web de la compañía.

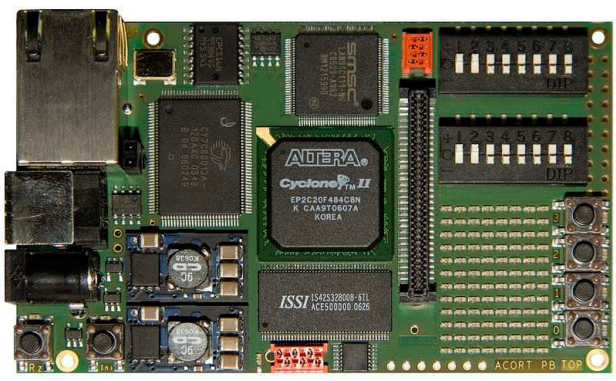
Esta tarjeta dispone de varias tarjetas de expansión entre las que cabe destacar FOXGM que permite añadir un modem GSM/GPRS a la tarjeta y de esa forma hacer que se comunique con nuestro teléfono móvil, por ejemplo.

Otra de las tarjetas de expansión disponibles para la Fox es Colibrí. Esta tarjeta añade una FPGA convirtiendo el tándem en un sencillo y potente sistema para introducirnos en el mundo del diseño hardware con dispositivos programables.

La FOXBoard original utilizaba un procesador RISC a 100 MHz y los puertos USB eran 1.1. La nueva versión de la tarjeta (FOX Board G20), se ha pasado a un procesador ARM9 y ha ampliado tanto los puertos USB a la especificación 2.0 como los puertos de expansión que proporciona.

FPGA4U

Acabamos de comentaros una opción para el desarrollo con FPGA libre, pero hay otra digna de mención, si bien, esta la tendréis que construir vosotros mismos, a no ser que os matriculéis en la EPFL en Suiza.



FPGA4U es un diseño orientado a la docencia y al desarrollo de prácticas en la universidad, así que, en principio, adecuado para introducirnos en este mundo.

La tarjeta está basada en una FPGA Altera Cyclone II EP2C20 y proporciona un interfaz Ethernet y otro USB, además de un array de LEDS y varios interruptores. Junto a estos periféricos, la tarjeta ofrece 32Mb de SDRAM (Synchronus DRAM).

OPEN GRAPHICS

Nos alejamos ya de los computadores, para introducirnos en otro tipo de hardware también libre, y comenzamos con el proyecto Open Graphics, un veterano en este mundo.

El objetivo del proyecto es el desarrollo de tarjetas gráfica abiertas. Como todos sabéis, las tarjetas gráficas se han convertido en los últimos años en el principal problema desde el punto de vista del hardware y el software libre.

Los drivers y librerías que permiten acceder a toda la potencia de las tarjetas gráficas son propietarios y por lo tanto cerrados y esta ha sido una de las razones que han impulsado este proyecto.

La primera tarjeta gráfica, conocida como OGD1 ya está disponible y se puede adquirir por unos 750 USD. Las características publicadas son:

- Salidas S-Video y Video Compuesto
- 1 Salida de video RGB analógico (resolución max.: 2048 X 1536 70Hz)
- 2 Salidas de DVI-I dual link (max resolution: 2560 X 1600 60Hz)
- 66 señales I/O de usuario en un conector Hirose 92-pines
- 68 señales I/O de usario en un conector IDC 100-pines
- Bus de memoria de 128 bits
- Diseño hardware Open Source
- Refrigeración pasiva

Una iniciativa excelente y un proyecto interesante para los que queráis profundizar en los sistemas gráficos y como funcionan las tarjetas de vídeo. Lamentablemente el precio y la disponibilidad de estas unidades no es muy conveniente por el momento.

USRP

Para terminar con nuestro limitado paseo por el mundo del hardware abierto vamos a hablaros del USRP (*Universal Software Radio Peripheral*), un proyecto para introducirnos en el mundo de los sistemas de radio comunicación de una forma "asequible" (entre 1000 y 2000 dolares) y libre :). Bueno, realmente, un sistema para introducirnos en el mundo de la Radio Software.

Una radio software es un sistema de radio comunicación en el que ciertos componentes se implementan en software, es decir, en un ordenador. Para ello es necesario digitalizar la señal de radio de forma que el ordenador pueda trabajar con ella.

Para poder hacer esto necesitamos que nuestra señal de radio esté a una frecuencia a la que un conversor analógico digital pueda capturarla y eso es lo que hace USRP (contado de forma sencilla). USRP se encarga de manejar la frecuencia intermedia y la cabeza de radio (lo que los ingleses llaman RF Front-end) y ofrecer un interfaz al ordenador para obtener los datos y controlar ese proceso.

USRP nos ofrece una plataforma abierta para experimentar con radios software.

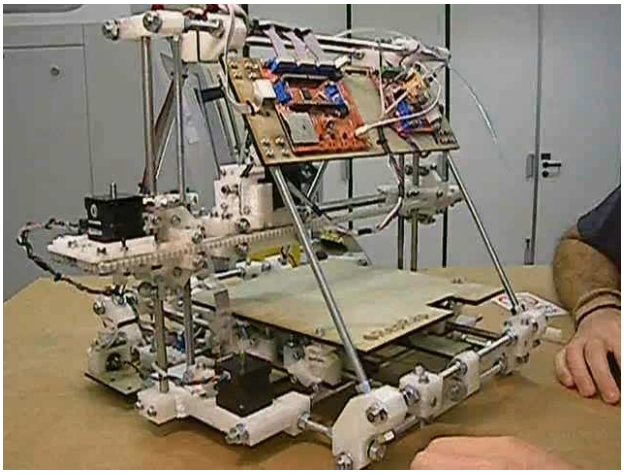
USPR proporciona un diseño modular para la cabeza de radio, permitiendo la instalación de distintas tarjetas de transmisión o recepción dependiendo de la señal con la que se desea trabajar.

En el otro extremo de la cadena está el ordenador, con el que se comunica utilizando un interfaz USB, para el caso de USRP, o un interfaz Gigabit Ethernet, en el caso del USRP2, que además ofrece conversores AD/DA de mayor frecuencia.

A partir de este punto el ordenador se encarga de extraer la información relevante. Para ello, USPR utiliza GNURadio un ambicioso proyecto para la implementación de radios software que nos proporciona los algoritmos de procesado de señal y los bloques que necesitamos para implementar estos sistemas en un PC normal y corriente.

IMPRESORAS 3D

El proyecto RepRap (*Replicating Rapid Prototiper* o prototipador rápido replicante ?¿) es una iniciativa para el desarrollo de una impresora 3D capaz de "imprimirse" a sí misma, y por lo tanto replicarse. Toda la información del proyecto se distribuye con licencias libres y hasta la fecha han producido dos modelos: "Darwin" y "Mendel".



Este tipo de impresoras 3D utilizan un sistema de ejes para mover un cabezal que deposita pequeñas gotas de plástico fundido hasta crear el modelo buscado.

Otra variante de este tipo de tecnología, quizás más popular, es la impresora CupCake de MakerBot Industries. Esta funciona de forma similar y está disponible comercialmente por unos 1000 dolares.

Aunque quizás un poco caras, la posibilidad de producir piezas de plástico (como cajas para prototipos, engranajes, etc...) para nuestros trabajos en nuestra propia casa, es un lujo que hasta no hace mucho no nos podíamos ni imaginar.

HASTA LA PRÓXIMA

Esperamos que os haya parecido interesante el concepto del hardware abierto y que os animéis a soportar estos proyectos que, en la humilde opinión del autor, tienen una importancia sobresaliente para que no perdamos el contacto con la tecnología que nos rodea y nos convirtamos en esclavos de esas máquinas simplemente por que no las conozcamos. ■

RECURSOS

Arduino:
http://www.arduino.cc/

BeagleBoard y Beagleboard-XM:
http://beagleboard.org

OpenPandora:
http://www.open-pandora.org/

OMBeagle:
http://projects.goldelico.com/p/ombeagle/

Wild Ducks:
https://thewildducks.wordpress.com/

PandaBoard:
http://pandaboard.org/

EAP GTA04:
http://www.handheld-linux.com/wiki.php?page=GTA04-Early-Adopter

FOX Board:
http://foxlx.acmesystems.it/?id=4

Colibri (FPGA para FOX Board):
http://eshop.acmesystems.it/?id=COLIBRI

FPGA4U:
http://fpga4u.epfl.ch/wiki/Main_Page

ODG1:
http://wiki.opengraphics.org/tiki-index.php

USRP:
http://www.ettus.com/order

GNURadio:
http://gnuradio.org/redmine/wiki/gnuradio/USRP

RepRap:
http://reprap.org/wiki/Main_Page

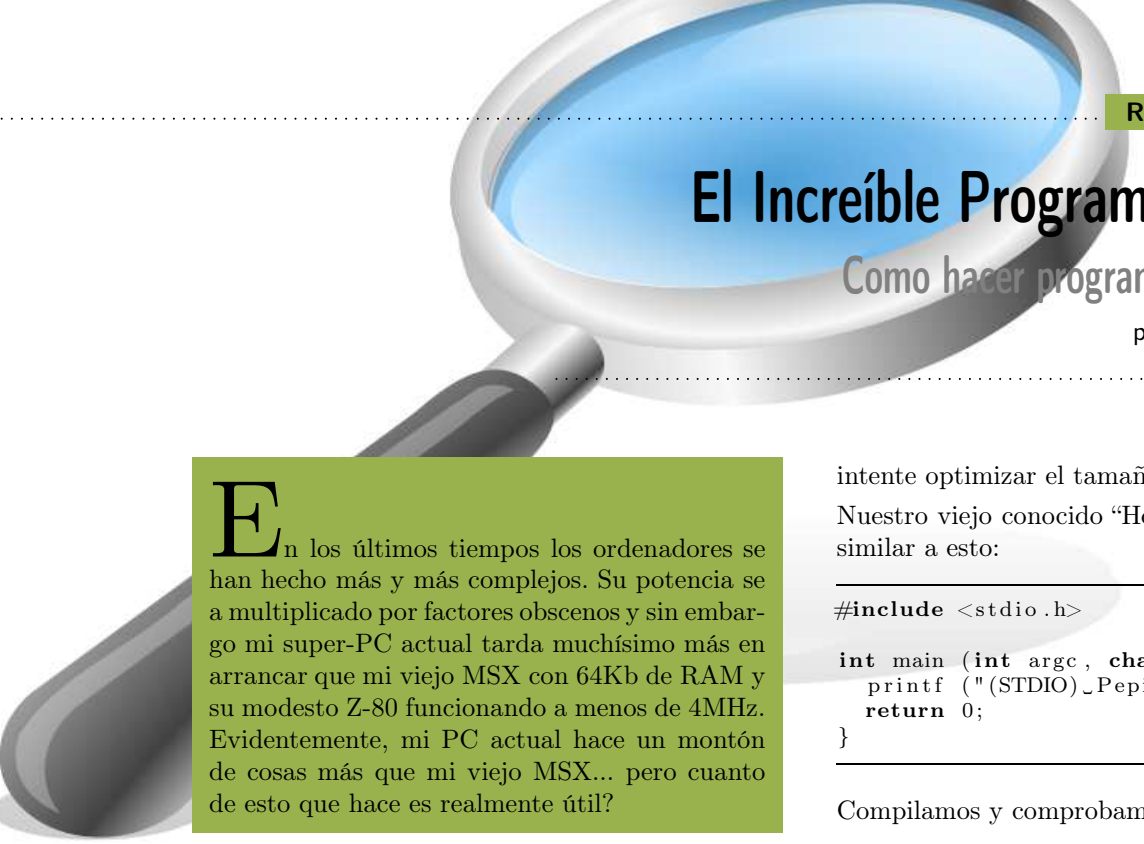
MakerBot:
http://makerbot.com/

DesignSomething:
http://designsomething.org/

El Increíble Programa Menguante

Como hacer programas muy pequeñitos

por Er Mangante Menguante



En los últimos tiempos los ordenadores se han hecho más y más complejos. Su potencia se a multiplicado por factores obscenos y sin embargo mi super-PC actual tarda muchísimo más en arrancar que mi viejo MSX con 64Kb de RAM y su modesto Z-80 funcionando a menos de 4MHz. Evidentemente, mi PC actual hace un montón de cosas más que mi viejo MSX... pero cuanto de esto que hace es realmente útil?

Los sistemas generalistas hacia los que evolucionó la informática doméstica en los 90 (vale, esto son los PC normales y corrientes) ha dado paso, en los albores del siglo XXI a sistemas mas específicos y modestos, cuya finalidad es realizar tareas concretas de forma eficiente, sencilla y sobre todo barata. Estos sistemas se han "re-bautizado" como sistemas embebidos, y si los comparamos con mi viejo MSX, siguen siendo infinitamente más potentes. Sin embargo puestos al lado de mi PC resultan mucho mas modestos.

El uso de GNU/Linux se ha popularizado en el diseño de estos sistemas embebido, fundamentalmente por el control que proporciona al desarrollador a la hora de personalizar el sistema. En este artículo vamos a comprobar cómo de pequeños pueden ser nuestros programas, ya que, en este tipo de sistemas, el tamaño sí importa.

Los denominados sistemas embebidos se han popularizado durante los últimos años

Para ello, vamos a tomar nuestro querido "Hola Mundo" e intentaremos reducirlo a su mínima expresión a través de las distintas posibilidades al alcance de nuestra mano.

REFERENCIA

A la hora de comparar cosas conviene tener una referencia. En nuestro ejemplo, tomaremos como referencia nuestra versión de "Hola Mundo" escrita en lenguaje C y compilada con el compilador de GNU. Puesto que estamos comparando tamaños, utilizaremos la opción de compilación `-Os` que le indica al compilar que intente optimizar el tamaño del código resultante.

Nuestro viejo conocido "Hola Mundo" en C sera algo similar a esto:

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  printf ("(STDIO) Pepita Pulgarcita!!!\n");
  return 0;
}
```

Compilamos y comprobamos el tamaño:

```
occams@razor $ gcc -Os -o hola-ref hola.c
occams@razor $ ls -l hola-ref
-rwxr-xr-x 1 occams razor 6.6K 2007-11-21 09:53 hello
```

Bueno, 6.6 Kb... no esta mal para un programa que solamente imprime un mensaje en pantalla.

HOLA MUNDO DESDE C++

C++ es uno de los lenguajes más utilizados en la actualidad. Una de las *pegas* que tiene este lenguaje, es que general ejecutables muy grandes. Vamos a comprobarlo con nuestro sencillo ejemplo:

```
#include <iostream>
using namespace std;

int main (int argc, char *argv[]) {
  cout << "(STDIO) Pepita Pulgarcita!!!" << endl;
  return 0;
}
```

Compilamos nuestra versión C++ y comprobamos su tamaño.

```
occams@razor $ g++ -Os -o hola-cpp hola.cpp
occams@razor $ ls -l
-rwxr-xr-x 1 occams razor 8.2K 2007-11-19 14:56 hola-cpp
```

Como era de esperar el tamaño es mayor que el de su versión C, pero tampoco parece algo exagerado... Bueno, me voy a guardar un par de detalles al más puro estilo Agatha Christie para sorprender más tarde :).

ELIMINANDO LO QUE SOBRA

A la hora de reducir el tamaño de un ejecutable hay ciertas partes de éstos que se pueden eliminar.

Lo primero que nos sobra, cuando nuestro programa este depurado, es la información de depuración. Esta información es generada por el compilador cuando utilizamos la opción `-g`. Como podéis observar, para nuestras pruebas no la hemos utilizado, pero a modo de ejemplo, veamos cuantos bytes implica esta información para un programa tan sencillo como nuestro "Hola Mundo".

```
occams@razor $ g++ -g -Os -o hm-cpp hola.cpp
occams@razor $ gcc -g -Os -o hm-ref hola.c
occams@razor $ ls -lh hola-cpp hola-ref
-rwxr-xr-x 1 occams razor  44K 2007-11-21 10:12 hm-cpp
-rwxr-xr-x 1 occams razor 7.3K 2007-11-21 10:12 hm-ref
```

La versión en C no crece demasiado, pero la versión C++ ocupa unas 5 veces más cuando se incluye información de depuración... así que recordad eliminar esta información en la versión final de vuestros proyectos o utilizar una "build tool" mínimamente decente.

La información de depuración tiene un gran impacto en el tamaño de los programas

Además de la información de depuración, hay algunas cosillas más que el compilador genera y que no son necesarias para que el programa se ejecute correctamente. La herramienta `strip` se encarga de eliminar esta otra información sobrante.

```
occams@razor $ strip hm-cpp hm-ref
occams@razor $ ls -lh hm-cpp hm-ref
-rwxr-xr-x 1 occams razor 4.1K 2007-11-21 10:16 hm-cpp
-rwxr-xr-x 1 occams razor 3.0K 2007-11-21 10:16 hm-ref
```

Sí, una de las cosas que strip elimina es la información de depuración :). Como podéis observar, el tamaño de nuestros programas se reduce a la mitad tras pasarlos por `strip`, pero esto es debido a que los programas son muy sencillos. No esperéis una reducción de tamaño del 50 % en el caso general.

ALTERNATIVAS A GLIBC

Hasta el momento, hemos obviado el hecho de que nuestras versiones del "HM" utiliza las librerías estándar C y C++ respectivamente. Es decir, estamos utilizando la función `printf` y la clase `cout` para imprimir nuestro mensaje. Estas librerías son los elementos principales que contribuyen al tamaño final de nuestro programa.

Afortunadamente, existen alternativas a estas librerías estándar que, sacrificando algunas funcionalidades, hacen que el tamaño de nuestros programas sea mucho menor y por lo tanto permiten ahorra ese preciado espacio de disco o memoria tan escaso en algunos sistemas.

Para nuestra comparación vamos a basarnos en dos de las librerías más utilizadas y conocidas en el mundo de los sistemas embebidos. Ambas son sustitutas de la librería C estandard, por lo que no nos servirán si nuestra aplicación está escrita en C++. Volveremos sobre esto un poco mas tarde. Por ahora vamos a generar nuestros nuevos ejecutables.

La primera librería que vamos a utilizar es uclib. Esta librería se distribuye como un '*toolchain*, es decir, como un conjunto de herramientas de desarrollo (compilador, linker, ensamblador, etc...). Una vez instalada (lo mas sencillo es utilizar el paquete que proporcione tu distribución) la forma de usarla es muy sencilla:

```
occams@razor $ i386-uclibc-linux-gcc -Os -o hm-uc hola.c
occams@razor $ ls -lh hm-uc
-rwxr-xr-x 1 occams razor 3.9K 2007-11-21 10:32 hm-uc
occams@razor $ strip hm-uc; ls -lh hm-uc
-rwxr-xr-x 1 occams razor 2.3K 2007-11-21 10:33 hm-uc
```

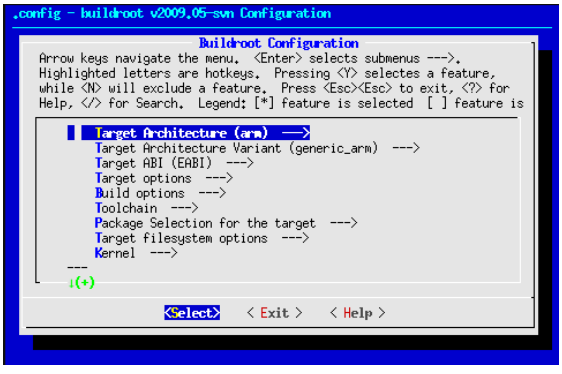
Bien, nos hemos puesto por debajo de los 3 Kb.

Veamos ahora que obtenemos con dietc, que aparentemente es incluso mas ligera que uclib. Dietc proporciona un sencillo *wrapper* para la compilación:

```
occams@razor $ diet -Os gcc -o hm-diet hola.c
occams@razor $ ls -lh hm-diet
-rwxr-xr-x 1 occams razor 2.0K 2007-11-21 10:36 hm-diet
occams@razor $ strip hm-diet; ls -lh hm-diet
-rwxr-xr-x 1 occams razor 804 2007-11-21 10:33 hm-diet
```

Wow!... poco más de 800 bytes. Esto ya es algo mas razonable para un programa que solo imprime un mensaje en la pantalla. El precio que hay que pagar por generar un ejecutable tan pequeño es cierta pérdida de funcionalidad, comparada con uclib y por su puesto con la librería C estándar de GNU, pero dependiendo de nuestra aplicación, muy probablemente no necesitemos esas funcionalidades que estamos sacrificando.

uclibc y DietC son alternativas ligeras a la librería C estándar



Menú de Configuración de ucLibC (Buildroot).

```
occams@razor $ file hola*
hm-ref:   ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.0,
          dynamically linked (uses shared libs), stripped
hm-cpp:   ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.0,
          dynamically linked (uses shared libs), stripped
hm-uclib: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
          dynamically linked (uses shared libs), stripped
hm-diet:  ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
          statically linked, stripped
```

Figura 1. Salida del comando file para nuestros ejemplos

TAMAÑOS REALES

Hemos visto el efecto que tienen las librerías estándar en el tamaño de los ejecutables, pero como decíamos al principio del articulo, nos hemos guardado un detalle bastante importante, al más puro estilo de las novelas de Agatha Christie.

Como todos sabéis, el comando `file`, nos da información sobre el tipo de fichero que le pasamos como parámetro. La figura 1 nos muestra esta información.

Vemos que todos han tenido un pequeño encuentro con `strip`, pero a excepción de la versión compilada con *dietc*, el resto de nuestros ejemplos están linkados dinámicamente, es decir, necesitamos una serie de librerías extra para poder ejecutar el programa. Por el contrario, `hm-diet` funcionara en cualquier sistema, independientemente de la versión de la librería C estándar instalada, siempre y cuando se trate de una plataforma x86 y un sistema operativo GNU/Linux.

Para hacernos una idea del espacio en disco y en memoria que necesitaremos para ejecutar cada uno de nuestros ejemplos, vamos a compilarlos estáticamente, de forma que las librerías utilizadas se incluyan en el ejecutable final (igual que nuestro `hm-diet`). Para ello, utilizaremos la opción de compilación `-static` y recompilaremos todos nuestros ejemplos:

```
occams@razor$ g++ -Os -static est-cpp hola.cpp
occams@razor$ gcc -Os -static est-ref hola.c
occams@razor$ i386-uclibc-linux-gcc -static -Os \
> -o est-uc hola.c
occams@razor$ ls -lg est*
-rwxr-xr-x 1 occams razor 1.2M 2007-11-21 11:45 est-cpp
-rwxr-xr-x 1 occams razor 525K 2007-11-21 11:46 est-ref
-rwxr-xr-x 1 occams razor  10K 2007-11-21 11:47 est-uc
occams@razor$ strip est*
occams@razor$ ls -lh est*
-rwxr-xr-x 1 occams razor 967K 2007-11-21 11:48 est-cpp
-rwxr-xr-x 1 occams razor 465K 2007-11-21 11:48 est-ref
-rwxr-xr-x 1 occams razor 5.4K 2007-11-21 11:48 est-uc
```

Compilando estáticamente nuestro programa comprobaremos tamaño real.

Bien, los resultados están bastante claros. La versión C++ requiere mucho más espacio en disco y memoria que la versión C equivalente. Por supuesto, esta diferencia se hace menor a medida que los programas son más grandes y complejos y, dependiendo del tipo de aplicación, las capacidades orientadas a objetos de C++ pueden justificar sobradamente el exceso de espacio requerido.

De la misma forma, si realmente no se necesitan esas funcionalidades especiales de C++, merece la pena pararse un momento a pensar si realmente, ese programa debe ser codificado en C++ o en C.

¿CÓMO LO HACEN?

Esta sea quizás la pregunta lógica tras los resultados que acabamos de obtener. Ya os adelantamos hace un momento que librerías como *uCLibc* o *DietC* no proporcionan la misma funcionalidad que la librería C estándar.

Sin embargo, tiene que haber algo más. En nuestra prueba, parece que la función `printf` ocupa unos 500 Kb... esa es mucha funcionalidad :).

Parece lógico pensar que 500 Kb contienen algo más que la función `printf`, y por lo tanto, que estas otras librerías son capaces de utilizar solo las funciones que el programa necesita.

Así que vamos a ver como consiguen hacer esto. Para ello, lo primero que necesitamos es una sencilla librería con la que hacer nuestras pruebas. Aquí tenéis el código de una *sencilla librería aritmética.*

```
int func1 (int a) {
  return a + 2;
}

int func2 (int a, int b) {
  return a + b;
}

int func3 (int a, int b) {
  return a - b;
}

int func4 (int a, int b) {
  return a * b;
}
```

Ahora solo tenemos que crear nuestra libería de la siguiente forma:

```
occams@razor$ gcc -c alib.c
occams@razor$ar -cvq libalib.a alib.o
```

Nuestro programa de ejemplo, va a utilizar solamente una de las funciones de nuestra sencilla librería. Aquí tenéis una posibilidad.

```
#include "alib.h"

int main (int argc, char *argv[]) {
  return func2 (5, 3);
}
```

El fichero `alib.h` estamos seguros que lo podéis crear vosotros mismos. Si tenéis algún problema solo tenéis que descargaron el código de este artículo de nuestra web.

Compilamos nuestro sencillo programa y echamos un ojo a ver si las funciones de nuestra librería andan por ahí.

```
occams@razor$ gcc -static -Os -o static01 \
> main.c libalib.a
occams@razor$ nm static01 | grep " func."
0804825c T func1
08048267 T func2
08048272 T func3
0804827f T func4
```

Ahí están las cuatro... pero a nosotros solo nos interesa `func2`...

INCLUYENDO SOLO LO QUE NECESITAMOS

Para decirle al linker que solo queremos incluir en nuestro ejecutable las funciones que realmente estamos utilizando, tenemos que compilar nuestra librería con algunas opciones especiales.

```
occams@razor$ gcc -fdata-sections \
> -ffunction-sections  -c alib.c -o alib1.o
occams@razor$ar -cvq libalib1.a alib1.
```

Estas opciones le dicen al compilador que genere una sección por cada bloque de datos y por cada función en el código. Y que es eso de las secciones. Os preguntaréis. A ello vamos.

Con las opciones de compilación adecuadas podemos incluir solo las funciones que utilizamos en nuestros programas.

Para poder ver las secciones de nuestras librerías utilizaremos la utilidad `readelf`. La salida de este programa para nuestra librería inicial es la siguiente.

```
occams@razor$ readelf -S libalib.a

File: libalib.a(alib.o)
There are 9 section headers, starting at offset 0xd4:

Section Headers:
  [Nr] Name              Type      Addr     Off    Size
  [ 0]                   NULL      00000000 000000 000000
  [ 1] .text             PROGBITS 00000000 000034 00002f
  [ 2] .data             PROGBITS 00000000 000064 000000
  [ 3] .bss              NOBITS   00000000 000064 000000
  [ 4] .comment          PROGBITS 00000000 000064 00002a
  [ 5] .note.GNU-stack PROGBITS 00000000 00008e 000000
  [ 6] .shstrtab         STRTAB   00000000 00008e 000045
  [ 7] .symtab           SYMTAB   00000000 00023c 0000b0
  [ 8] .strtab           STRTAB   00000000 0002ec 000020
```

Como podéis ver hay un montón de información aquí. Para el tema que nos ocupa, la sección que nos interesa es `.text`. Esta sección, es en la que el compilador pone el código ejecutable de nuestra librería.

Veamos, antes de continuar, que nos dice `readelf` sobre nuestra segunda librería, la que hemos compilador con los flags especiales.

```
occams@razor$ readelf -S libalib1.a

File: libalib1.a(alib1.o)
There are 13 section headers, starting at offset 0x104:

Section Headers:
  [Nr] Name              Type      Addr     Off    Size
  [ 0]                   NULL      00000000 000000 000000
  [ 1] .text             PROGBITS 00000000 000034 00000b
  [ 2] .data             PROGBITS 00000000 000034 000000
  [ 3] .bss              NOBITS   00000000 000034 000000
  [ 4] .text.func1       PROGBITS 00000000 000034 00000b
  [ 5] .text.func2       PROGBITS 00000000 00003f 00000b
  [ 6] .text.func3       PROGBITS 00000000 00004a 00000d
  [ 7] .text.func4       PROGBITS 00000000 000057 00000c
  [ 8] .comment          PROGBITS 00000000 000063 00002a
  [ 9] .note.GNU-stack PROGBITS 00000000 00008d 000000
  [10] .shstrtab         STRTAB   00000000 00008d 000075
  [11] .symtab           SYMTAB   00000000 00030c 0000f0
  [12] .strtab           STRTAB   00000000 0003fc 000020
```

Vemos que el tamaño de nuestra sección `.text` es ahora cero, y que tenemos cuatro secciones adicionales... una por cada una de las funciones que hemos definido en nuestra libería.

Esta es la información que el linker va a utilizar para incluir solo las secciones de interés en nuestro programa estático.

Ahora ya podemos recompilar nuestro programa y generar un ejecutable solo con las funciones que realmente necesitamos. Una vez más necesitamos una opción de compilación especial, bueno, realmente una opción de linkado... La opción que le dice al linker que ignore las secciones que no son utilizadas.

```
occams@razor$ gcc -static -Wl,--gc-sections main.c \
> libalib1.a -o static03
occams@razor$ nm static03 | grep " func."
080481de T func2
```

Sí, el flag `-Wl,` nos permite pasar opciones directamente al linker, y la opción `--gc-sections` es la que elimina lo que sobra (gc viene de *Garbage Collection* o recolección de basura).

Si comprobáis el tamaño del nuevo ejecutable, veréis que es ligeramente más pequeño. La diferencia no es mucha, ya que lo gordo viene de la librería C que no hemos compilado para producir secciones por cada función.

CONTROLANDO LAS SECCIONES

Acabamos de ver como crear, de forma automática, una sección por cada función de nuestra librería de forma que, en la fase de enlazado, podamos eliminar las secciones (y por lo tanto funciones) que no utilizamos, reduciendo el tamaño de nuestro ejecutable estático.

Controlar las secciones de nuestro programa, tiene otras ventajas. Por ejemplo, nos permite agrupar funciones relacionadas en una misma zona de la memoria mejorando la localidad de código que se ejecutar, normalmente junto.

Para poder alcanzar ese control, tenemos que meternos en el escabroso mundo de como el compilador genera el código. Si no se hace con cuidado el código acabará dependiendo de un compilador en concreto y muy probablemente de una versión especial de ese compilador. La solución. Relativamente complejos ficheros de cabecera haciendo uso de esos *defines* implícitos que cada compilador añade a su libre albedrío.

Dicho esto, aquí tenéis un ejemplo de como poner las funciones que queramos en la sección que nos apetezca.

```
__attribute__((section(".occams1")))
int func1 (int a) {
  return a + 2;
}

__attribute__((section(".occams1")))
int func2 (int a, int b) {
  return a + b;
}

__attribute__((section(".occams2")))
int func3 (int a, int b) {
  return a - b;
}

__attribute__((section(".occams2")))
int func4 (int a, int b) {
  return a * b;
}
```

Llamemos a este fichero `blib.c` y repitamos el proceso anterior para ver que obtenemos.

```
occams@razor$ gcc -c blib.c -o blib.o
occams@razor$ ar -cvq libblib.a blib.o
occams@razor$ readelf -S libblib.a
File: libblib.a(blib.o)
There are 11 section headers, starting at offset 0xe4:

Section Headers:
  [Nr] Name              Type       Addr     Off    Size
  [ 0]                   NULL       00000000 000000 000000
  [ 1] .text             PROGBITS   00000000 000034 000000
  [ 2] .data             PROGBITS   00000000 000034 000000
  [ 3] .bss              NOBITS     00000000 000034 000000
  [ 4] .occams1          PROGBITS   00000000 000034 000016
  [ 5] .occams2          PROGBITS   00000000 00004a 000019
  [ 6] .comment          PROGBITS   00000000 000063 00002a
  [ 7] .note.GNU-stack PROGBITS   00000000 00008d 000000
  [ 8] .shstrtab         STRTAB     00000000 00008d 000057
  [ 9] .symtab           SYMTAB     00000000 00029c 0000d0
  [10] .strtab           STRTAB     00000000 00036c 000020
(...)
occams@razor$ gcc -static -Wl,--gc-sections main.c \
> libblib.a -o static04
occams@razor$ nm static04 | grep " func."
0809e189 T func1
0809e194 T func2
```

Ahora `readelf` muestra las dos nuevas secciones que hemos creador utilizando la directiva del compilador `__attribute:__`. Como era de esperar, con nuestra nueva agrupación de funciones en secciones, nuestro ejecutable final contendrá también `func2` aún cuando no la utiliza.

Observad también como el tamaño de la sección `.text` es ahora cero.

OTRAS OPCIONES

En este artículo os hemos hablado de `dietC` y `ulibc` pero existen, al menos, otras dos alternativas para hacer nuestros programas más pequeños, dignas de mención.

La primera de ellas es `newlib` más genérica desde el punto de vista del sistema operativo. Esta librería es utilizada en distribuciones comerciales del GCC.

La segunda es `EGLIBC` (*Embedded GLibC*). Este proyecto busca dar soporte a sistemas embebidos, manteniendo un alto nivel de compatibilidad con GlibC (la librería C estándar de GNU). Quizás su principal característica es su capacidad de configuración de los componentes de la misma.

Finalmente comentaros que existe un proyecto para desarrollar una librería similar a las que hemos comentado para aplicaciones escritas en C++. Su nombre es `uClibC++` :).

En el cuadro de recursos podéis encontrar algunas referencias para saber más sobre estas alternativas.

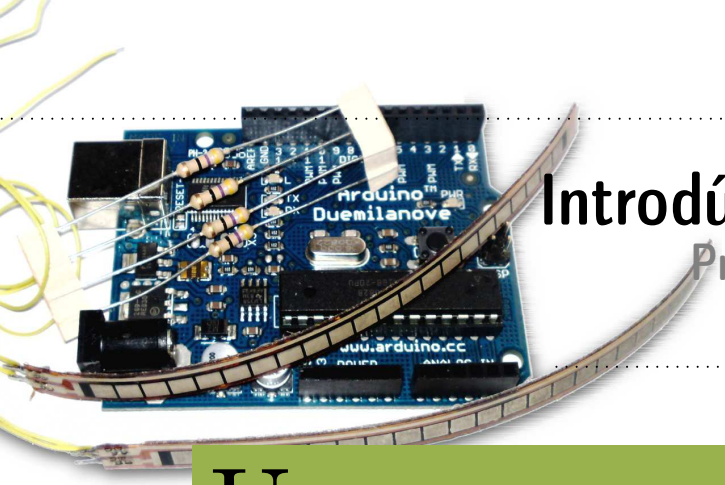
Hasta el próximo número.

RECURSOS

ALTERNATIVAS LIBRERÍA C

dietlibc. http://www.fefe.de/dietlibc/
uClibC. http://www.uclibc.org/
Newlib. http://www.sourceware.org/newlib/
EGLIBC. http://www.eglibc.org/home
uClibC++. http://cxx.uclibc.org/

Interesante tutorial sobre como crear ejecutables verdaderamente pequeños.
A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux
http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html



Introdúcete en la Realidad Virtual

Primeros pasos para un DataGlove casero

por er Cortaó de Yervas

Un dataglove, o para nosotros, un guante de realidad virtual, es uno de los emblemas, junto con los HMD (*Head Mounted Displays* o Pantallas Montadas en la cabeza :) de los sistemas de realidad virtual. Si bien su utilidad es cuestionable, dependiendo de la aplicación, vamos a contaros las bases para construir uno de estos vosotros mismos.

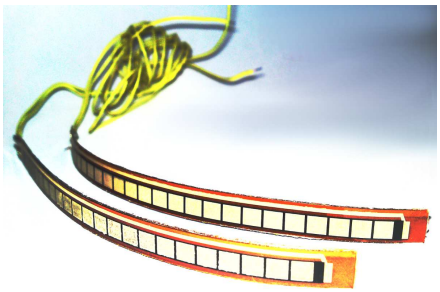
En este primer artículo vamos a presentar la toma de contacto con los distintos elementos que utilizaremos en este miniproyecto. Sin más preámbulos, la idea es la de construir un sencillo *dataglove* que utilizar como elemento de interfaz con nuestro ordenador.

Con un Arduino y unos cuantos sensores flexibles podremos construir un rudimentario dataglove

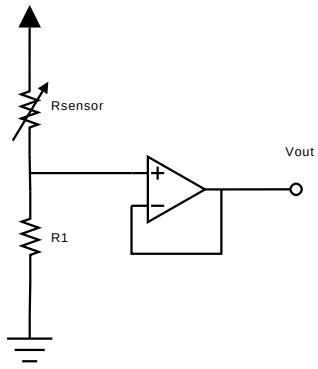
Para ellos vamos a utilizar un Arduino, unos cuantos sensores flexibles y un acelerómetro. El Arduino Duemilanove, nos ofrece 6 entradas analógicas, así que para poder capturar los 3 ejes de nuestro acelerómetro vamos a restringir nuestro guante a solo tres dedos, en lugar de los cinco habituales.

POSICIÓN DE LOS DEDOS

Como adelantábamos, para determinar la posición de los dedos, vamos a utilizar sensores flexibles. Básicamente se trata de un sensor cuya resistencia varía cuando se dobla. Nuestros sensores tienen una resistencia de unos 10K en reposo y cuando se dobla el rango varia de 60K a 110K.



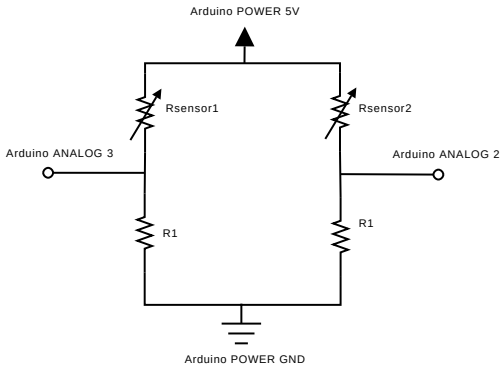
El datasheet del sensor ofrece información sobre como conectar el sensor en distintas circunstancias. Para nuestra prueba de concepto hemos elegido la más sencilla, un simple divisor de voltaje. El datasheet recomienda en uso de un amplificador operacional en modo seguidor o, lo que es lo mismo, como buffer de impedancia. El circuito sería algo como esto.



En esta primera fase hemos prescindido del amplificador operacional como sugiere el datasheet, y hemos conectado el pin V_{out} del diagrama directamente a una de las entradas analógicas del Arduino.

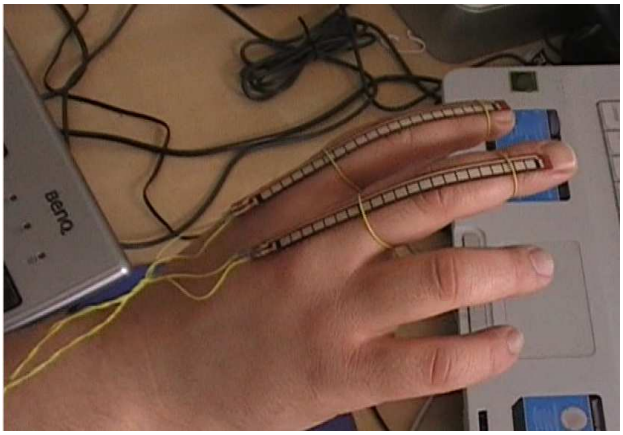
CONECTANDO LOS SENSORES

Aunque esto probablemente sea evidente para todos vosotros, incluimos un pequeño diagrama de como conectar todos los elementos de nuestro montaje. Aquí está:



Todas las conexiones etiquetadas como Arduino en la figura, se encuentra en el lado del conector de alimentación de la placa. El bloque `POWER` es el que se encuentra más cerca del conector y el bloque `ANALOG` el que se encuentra más lejos. Vamos que no tiene pérdida.

Una vez que todo está listo, nuestro pequeño engendro se parecerá a esto:



Bueno, la versión final habrá que integrarla con un guante de verdad para facilitar su uso. De la misma forma, para nuestra primera prueba hemos utilizado una placa de prototipos que habrá que sustituir por algo más sencillo que podamos esconder en nuestro guante VR.

LEYENDO DATOS

Con el sencillo esquema que acabamos de mostrar estamos en condiciones de escribir un primer *Sketch* para nuestro Arduino. En esta primera fase vamos a utilizar solamente dos sensores.

```
int potPin = 2;
int potPin1 = 3;

void setup()  {
  Serial.begin(9600);
}

void loop()  {
 int val, val1;

 while (1) {
  val = analogRead (potPin);
  val1 = analogRead (potPin1);

  delay(60);
  Serial.print("0:"); Serial.print(val)
  Serial.print(":1:");Serial.print(val1);
  Serial.print("\n");
 }
}
```

El programa es muy sencillo. Una vez inicializado el puerto serie en la función `setup`, entramos en un bucle infinito en el que leemos los datos de los puertos análogicos 2 y 3 (los que hemos elegido para este ejemplo).

Esos datos los enviamos a través del puerto serie con el siguiente formato.

`0:ValorSensor0:1:ValorSensor1`

Compilamos y cargamos nuestro programa en el Arduino.

Ahora no tenemos más que abrir el *Serial Monitor* para empezar a ver los datos que nos envía nuestro sensor a través del puerto serie virtual que nos ofrece Arduino.

En el monitor deberíamos ver algo como esto:

```
0:600:1:400
0:600:1:400
```

Los números variarán según la curvatura de los sensores.

ADQUIRIENDO DATOS DESDE C

La consola serie del entorno del Arduino es muy útil para la depuración de nuestros *Sketches* pero, a no ser que ese entorno sea suficiente para nuestra aplicación, tendremos que poder acceder a esos datos que nos envía Arduino desde nuestra propia aplicación.

Como hemos dicho un poco más arriba, Arduino se presenta al sistema operativo como un interfaz serie (a través de un puerto USB), así que la forma de comunicarnos con él es como con cualquier otro dispositivo serie.

Para nuestro ejemplo, hemos adaptado un sencillo programa desarrollado por Tod E. Kurt, quien gestiona un estupendo sitio web sobre Arduino con excelente información.

El entorno de desarrollo del Arduino nos permite adquirir señales análogicas de forma muy sencilla

Con esa aplicación (consultad el cuadro de recursos al final de artículo), disponemos del código básico para poder leer la posición de los dedos de nuestro simple prototipo.

UNA DEMO

La aplicación de Tod E. Kurt, solamente permite leer una línea del puerto serie de cada vez, para crear nuestra demo la hemos modificado añadiendo un bucle infinito en el que leemos los datos y los parseamos continuamente.

El código lo podéis encontrar en nuestra web, pero en esta sección os contamos que modificaciones fueron necesarias para hacer funcionar nuestra sencilla demo.

La primera fue la eliminación del flag `O_NDELAY` en la función `serialport_init` para hacer nuestro descriptor bloqueante.

La siguiente fue añadir tras la interpretación de los argumentos del programa un bucle principal que lee los datos del Arduino, los transforma y los reenvía a la salida estándar. Este esquema nos permite *empipar* estos datos a cualquier otro programa o incluso a través de la red con utilidades como netcat o nuestro querido NetKitty.

Este bucle principal será algo como esto:

```
while (1)
 {
   memset (buf, 0, 256);
   if (!serialport_read_until (fd,buf, '\n'))
     {
       buf[strlen (buf) - 1] = 0;
       sscanf (buf, "0:%d:1:%d", &v[0], &v[1]);
       if (v[0] > max[0]) max[0] = v[0];
       if (v[0] < min[0]) min[0] = v[0];

       if (v[1] > max[1]) max[1] = v[1];
       if (v[1] < min[1]) min[1] = v[1];

       r[0] = xform (v[0], min[0], max[0]);
       r[1] = xform (v[1], min[1], max[1]);

       printf ("ROT %f %f 0.0\n", r[0], r[1]);
       fflush (0);
       usleep (10);
     }
}
```

La primera parte del bucle determina los valores máximos y mínimos, de forma que tengamos información sobre el rango dinámico de los sensores. Dependiendo de la longitud de los dedos del usuario o como han sido colocados los sensores, estos valores máximos y mínimos que leeremos pueden variar bastante.

Podemos considerar esto como el sistema de calibración más tonto posible. Al arrancar la aplicación deberemos estirar y encoger los dedos para calcular el rango dinámico correcto.

Esto significa que los primeros valores que obtendremos de nuestro guante serán incorrectos.

Abriendo y cerrando la mano al iniciar la aplicación conseguimos calibrar el sistema

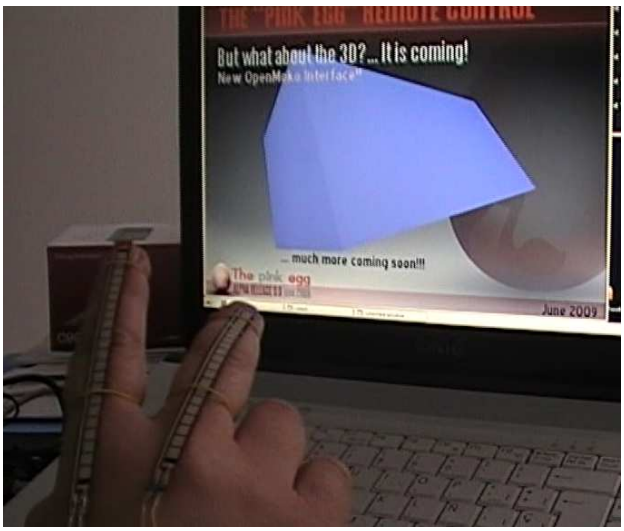
Luego llamamos a una función `xform`. Esta función no la incluimos aquí ya que en nuestra aplicación generaba datos adaptados para nuestra demo que probablemente no os sean de ninguna utilidad. En el caso más sencillo esta función puede producir un valor del sensor normalizado (entre 0 y 1), de acuerdo al rango dinámico actual (que le pasamos como parámetro).

LOS RESULTADOS

El programa que acabamos de comentar, imprime líneas con el formato `ROT a1 a2 0.0`, donde los valores `a1` y `a2` son dos ángulos proporcionales a la curvatura de cada uno de nuestros sensores. Eso es lo que calcula la función `xform`

Para completar la demo, hemos utilizado un sencillo programa que visualiza un cubo. Bueno, en realidad es un programa para realizar presentaciones que incluyen modelos 3D... pero también se puede utilizar para visualizar un cubo :).

Este programa es capaz de interpretar el comando que recibe a través de la entrada estándar, y además uno de los comandos que interpreta es `ROT`... casualidad?... yo no lo creo :).



Este comando rota el objeto 3D seleccionado en ese momento en cada uno de los ejes en función de los parámetros que acompañan al comando `ROT`.

Así que solo tenemos que conectar nuestro programa que toma datos del Arduino y los imprime por pantalla y conectarlo con nuestro supervisualizador de cubos, utilizando netcat o NetKitty.

El resultado es una bonita demo en la que moviendo dos de nuestros dedos podemos rotar un modelo 3D (un cubo en este caso) en torno al eje X o el eje Y.

Observad que estamos rotando un cubo, porque eso es lo que teníamos a mano y lo más sencillo. Queríamos probar el guante no desarrollar una aplicación. Lo importante es que somos capaces de enviar la información de como de curvado está cada uno de los sensores a nuestros programas.

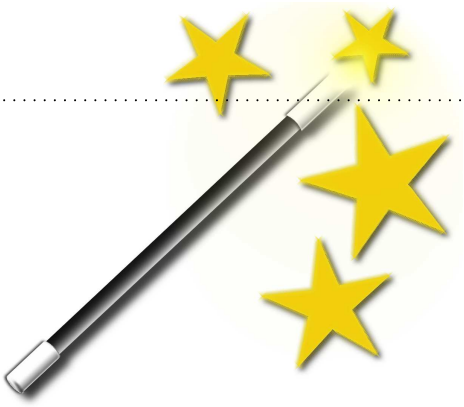
Cambiar el cubo por una mano articulada es un problema de programación gráfica independiente de como funcione nuestro guante.

CONCLUSION

Para concluir, con un Arduino (unos 25 EUR), unos cuantos sensores flexibles (el precio puede variar entre 8 y 13 USD) y unos cuantos componentes (resistencias y op-amps opcionales), podemos construir por menos de 100 EURs un *dataglove* con el que explorar el mundo de la realidad virtual. ■

RECURSOS

Código C para leer datos de Arduino
http://todbot.com/blog/2006/12/06/arduino-serial-c-code-to-talk-to-arduino/
Video del prototipo funcionando
http://www.youtube.com/watch?v=SHHILSQaSaU&feature=mfu_in_order&list=UL
Sensores Flexibles y Datasheet
http://www.sparkfun.com/products/8606



Con un par... de líneas

Chuletillas para hacer cosas mú rápido

por Tamariz el de la Perdíz

SCREENSHOTS CHULOS EN GNOME

GNOME viene con un montón de utilidades que no son muy conocidas. Una de ellas es `gnome-screenshot`, una aplicación que nos permite hacer capturas de pantalla.

Una de las cosas que nos permite esta utilidad es la de añadir una bonita sombra a nuestras capturas automáticamente. La línea de comandos sería más o menos esta:

```
gnome-screenshot -w -d 2 -e shadow
```

Este comando esperará dos segundos antes de capturar la ventana que tenga el focus como un fichero png con una sombra transparente alrededor.

Si no queréis escribir todas esas opciones siempre podéis utilizar la opción `-i` que os proporcionará un cómodo interfaz gráfico para elegir todos esos parámetros.

CAMBIAR EL WALLPAPER DE GNOME

En la misma línea del truco anterior, la utilidad `gconftool` nos permite acceder desde la línea de comando a varios de los parámetros de configuración del escritorio.

A modo de ejemplo, podemos cambiar la imagen de fondo del escritorio con el siguiente comando.

```
gconftool-2 --type string \
            --set /desktop/background/picture_filename \
            imagen.jpg
```

Donde `imagen.jpg` es la imagen que queremos poner como fondo de escritorio.

Por ejemplo, un sencillo script para descargar una imagen de cualquier web y ponerla como fondo de escritorio sería algo tal que así:

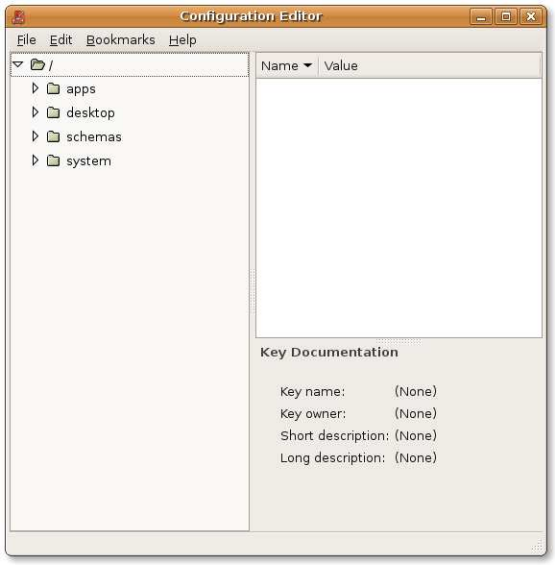
```
#!/bin/sh

while (true) do
    wget $1 -O ${HOME}/imagen.jpg
    gconftool-2 --type string \
                --set /desktop/background/picture_filename \
                imagen.jpg
    sleep 60  # Espera un minuto y repite
 done
```

Sí, todos los browsers tienen la opción de poner una imagen como fondo de escritorio...

CONFIGURACIÓN GNOME

La utilidad `gconf-edit` os permitirá acceder a todas esas opciones de GNOME que pensabais que no podíais cambiar.



Por supuesto, este screenshot se ha tomado con el primero de los trucos... de esta sección :)

Con esta herramienta podréis cambiar cosas como la posición de los botones en las ventanas, eliminar esos iconos del escritorio que no se pueden borrar y mucho más.

Antes de poneros a jugar con la herramienta os recomendamos hacer una copia de seguridad con el comando:

```
gconftool-2 --dump / > ~/gconf_backup.xml
```

Podéis restaurar vuestro backup con el siguiente comando:

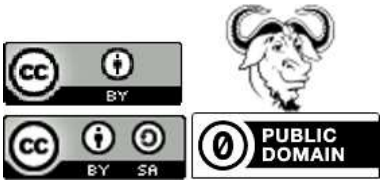
```
gconftool-2 --load ~/gconf_backup.xml /
```

Otra utilidad interesante de esto es la configuración del escritorio de máquinas recién instaladas.

Envía tus trucos

Puedes enviarnos esos trucos que usas a diario para compartirlos con el resto de lectores a la dirección:

`occams-razor@uvigo.es`



LICENCIAS DE LAS IMÁGENES EN ESTE NÚMERO

Gracias a todos los que comparten su trabajo!

por The Occam's Razor Team

EN LA PRÁCTICA

En el artículo "Open Hardware" en la sección "En la práctica", hemos utilizado las siguientes imágenes:

OpenMoko FreeRunner

Licencia: Creative Commons Attribution-Share Alike 3.0 Unported

Autor: OpenMoko Web Site (http://openmoko.com)

RepRap

Licencia: Creative Commons Attribution-Share Alike 3.0 Unported

Autor: CharlesC (CharlesC)

OpenPandpora

Licencia: Creative Commons Attribution 1.0 Generic

Autor: Michael Mrozek

Beagleboard-XM

Licencia: Creative Commons Attribution-Share Alike 3.0 license

Autor: BeagleBoard Web Site (http://beagleboard.org)

Pandaboard

Licencia: Creative Commons Attribution-Share Alike 3.0 license

Autor: Pandaboard Web Site (http://pandaboard.org)

Fox Board

Licencia: Creative Commons Attribution-ShareAlike 3.0 License

Autor: ACME Systems Web Site http://www.acmesystems.it/articles/00060/FOXLX416.jpg

FPGA4U

Licencia: GNU Free Documentation License 1.2.

Autor: FPGA4U Wiki Web Site (http://fpga4u.epfl.ch/wiki/Main_Page)

Open Graphics

Licencia: GPL Autor: Open Graphics Project: http://www.opengraphics.org/

USRP

Licencia: Dominio Público

Autor: Polvi

Arduino

Licencia: Dominio Público

Autor: H0dges

OTROS ARTÍCULOS

Logo OSG artículo sección "Ratas de Biblioteca"

Autor: Openscenegraph Web Site (http://www.openscenegraph.org/)

Gracias al proyecto OpenClipart (http://www.openclipart.org/) for sus estupendas imágenes cedidas al dominio público. Hemos utilizado las siguientes imágenes de este excelente sitio.

Fantasma artículo sección "Mú Rápido"

Autor: lemmling (http://www.openclipart.org/user-detail/lemmling)

Lupa artículo reverso tenebroso

Autor: The Structorr (http://www.openclipart.org/user-detail/TheStructorr)

OCCAM'S RAZOR

Porque lo más sencillo es lo más probable

nº1

nº2

nº3

nº4

nº5

NO TE CORTES CON LA NAVAJA DE OCCAM

descarga todos los números en http://webs.uvigo.es/occams-razor