

# Compilando y Corriendo TORO

*Audiencia: para programadores.*

Por Matías E. Vara

Ultima modificación: 27 / 02 / 2008

## Sobre este Papel :

Este simple texto explica como debe ser compilada la RTL de **Toro** y la aplicación de usuario, junto con las herramientas necesarias para su compilación.

Se explica como puede ser probado el sistema utilizando emuladores y los procedimientos de debug que brinda **Toro**.

También se trata de introducir la idea de compilar la aplicación de usuario junto al kernel en un único ejecutable.

Si hay algún link roto o algún comentario por favor contáctese conmigo.

Un especial saludo a mi amigo **KW**,

Matías E. Vara,  
[Toro.sourceforge.net](http://Toro.sourceforge.net)  
[matiasvara@yahoo.com](mailto:matiasvara@yahoo.com) .

# Contenido

Compilacion del kernel junto a la aplicación de usuario .....	4
Prueba del Sistema .....	6
Debug .....	8

## Compilación del kernel junto a la aplicación de usuario:

La compilación de la rtl es muy sencilla, como primer paso debe bajarse el paquete correspondiente a la arquitectura. Hasta el momento, los últimos paquetes son:

**Toro32-0.03-src**: que corresponde a procesadores i386.

**Toro64-0.03-src**: que corresponde a procesadores AMD x86-64.

Cuando se descomprime el paquete los directorios son los siguientes:

*/boot:*

Contiene los archivos de inicialización del sistema y la aplicación *build.exe* encargada de generar la imagen booteable de **Toro**.

*/rtl:*

Aquí se encuentra la RTL mínima, las unidades del kernel y los drivers.

*/:*

En la raíz se encuentra el archivo *Compile.cmd* encargado de la compilación de la RTL de **Toro** + la aplicación de usuario (en este caso *toro.lpr*).

.La estructura general de la aplicación de usuario deberá ser:

```
program toro;
```

```
{ $IFDEF FPC }  
{ $mode objfpc }  
{ $ENDIF }
```

```
uses Kernel in '..\rtl\Kernel.pas',  
    Process in '..\rtl\Process.pas',  
    Printk in '..\rtl\Printk.pas',  
    Memory in '..\rtl\Memory.pas',  
    Errno in '..\rtl\Errno.pas',  
    Debug in '..\rtl\Debug.pas',  
    Arch in '..\rtl\i386\Arch.pas'; // este es para caso de procesadores i386
```

```
Begin  
// Some Code  
End.
```

Para realizar la compilación diríjase al directorio de **Toro** y ejecute:  
> *Compile.cmd*

Si la compilación fue correcta se habrá compilado la RTL mínima, las unidades del kernel y la aplicación de usuario, y se habrá generado el archivo *toro.exe* .

Los paquetes incluyen una aplicación de usuario en forma de ejemplo en el archivo *toro.lpr* que se compila utilizando el procedimiento antes descrito.

Sobre 64 bits la compilación de **Toro** puede realizarse simplemente desde **Lazarus**.

**Lazarus** es una Interfaz grafica para el desarrollo de aplicaciones en lenguaje **Pascal** similar a **Delphi**, se distribuye bajo la licencia GPL, la url del proyecto es :

<http://www.lazarus.freepascal.org/>

Una vez instalada la interfaz debe ser abierto el archivo *toro.lpi* en el directorio de **Toro**, luego diríjase a *Ejecutar > Compilado Rápido*. Esto generara el archivo *toro.exe* en el directorio de **Toro**.

Las herramientas necesarias para compilar en 32 bits son:

- *FreePascal Compiler de 32 bits v. 2.0.2* en la url:

<http://www.freepascal.org/>

- *NASM v. 0.98.39* de la url:

<http://downloads.sourceforge.net/nasm/nasm-0.98.39-win32.zip?>

Esta utilidad es solo necesaria para la compilación de las herramientas de boot las cuales ya viene compiladas por defecto .

Para Windows de 64 bits:

- *FreePascal Compiler for 64 bits v. 2.2.1* y *Lazarus v. 0.9.24*:

[http://downloads.sourceforge.net/lazarus/lazarus-0.9.24-fpc-2.2.0-20071108-win64.exe?modtime=1194526686&big\\_mirror=1](http://downloads.sourceforge.net/lazarus/lazarus-0.9.24-fpc-2.2.0-20071108-win64.exe?modtime=1194526686&big_mirror=1)

Este paquete incluye los binarios de **Freepascal y Lazarus**.

- *Yasm v. 0.5.0 for Win64* de la url:

<http://www.tortall.net/projects/yasm/releases/yasm-0.5.0-win64.exe>

De nuevo, esta herramienta es solo necesaria para la compilación de las herramientas de boot.

## **ACLARACION:**

Los archivos *Compile.cmd* definen la variable de entorno:

*Set path=C:\FPC\2.0.2\bin\i386-Win32*

Se deberá modificar *path* colocando el directorio donde se encuentren todos los ejecutables necesarios para la compilación

## Prueba del Sistema

Una vez obtenida el archivo *toro.exe* debera generarse la imagen booteable.

Esto puede realizarse de dos maneras :

1 – Dirijase al directorio de **Toro** y ejecute :

> *RunOnQemu.cmd*

Esta aplicación crea la imagen booteable y emula **Toro** utilizando el emulador **Qemu** .

2 – Sobre **Lazarus** dirijase a :

*Ejecutar > Ejecutar*

Esto compila **Toro** y ejecuta *RunOnQemu.cmd* directamente .

Podra utilizarse cualquier emulador de **i386** o **AMD x86-64**, como por ejemplo **Bochs**, **Vmware**, etc.

Particularmente **Toro** trae los archivos necesarios para ser emulado sobre **Qemu**, tanto sobre 32 como 64 bits. También posee soporte para SMP. (Multiprocesamiento Simétrico) para hasta 255 procesadores.

.El archivo *RunOnQemu.cmd*, posee la línea de comandos para crear una maquina virtual sobre **Qemu**.

*Qemu v.0.9.1 for Windows* puede ser bajado desde :

<http://www1.interq.or.jp/t-takeda/qemu/qemu-0.9.1-windows.zip>

Incluye las versiones para 32 y 64 bits.

Para el correcto funcionamiento debera ser instalado en *C:\qemu* o debera editarse *RunOnQemu.cmd* para colocar la ruta correcta.

## Utilizacion de un sistema de Archivo en Qemu :

Sobre **Qemu** pueden agregarse hasta 4 disco identificados por :  
*hda* y *hdb* corresponden a la Controladora IDE Maestra , los discos maestro y esclavo , respectivamente .

*hdc* y *hdd* corresponden a la Controladora IDE Secundaria, los discos maestro y esclavo, respectivamente .

Para hacerlo debe agregarse a la linea de comando que llama a

**Qemu** -en el archivo *RunOnQemu.cmd*- la sentencia :

*-hda image.img*

Esto situa a la imagen de un disco duro del archivo *image.img* sobre la Controladora Maestro , disco maestro , en la maquina emulada .

Puede bajarse un Sistema de Archivo de Ejemplo de la url :  
<http://prdownloads.sourceforge.net/toro/LinuxRedHat.rar?download>  
descomprimalo en el directorio de **Toro**.

Esta es una imagen de un sistema **Linux RedHat** instalado en una particion EXT2.

## Utilizacion de una Interfaz de Red sobre Qemu :

Apartir de **Toro 0.03** tiene soporte para tarjetas de red ethernet compatibles con el modelo ne2000 .

Para simular una Red Virtual sobre Windows se utiliza la aplicacion **OpenVPN** , para mas informacion dirijase a <http://openvpn.net/> esta se distribuye bajo la licencia GPL.

*OpenVPN v. 2.0.9* puede ser bajado de :  
<http://openvpn.net/release/openvpn-2.0.9-install.exe>

Una ves bajado e instalado , simplemente debe agregarse a la linea de comando que llama a **Qemu** -en el archivo *RunOnQemu.cmd*- la sentencia:

```
-net nic,model=ne2k_pci -net nic,macaddr=00:87:87:87:87:87 -net tap,ifname=NOMBRE_DEL_ADAPTADOR
```

Esto emula una tarjeta de red ethernet ne2000, con la MAC dada ,utilizando el adaptador virtual dado en *ifname* .

Puede leerse un excelente tutorial que describe mas profundamente este procedimiento en la pagina :

<http://www.h7.dion.ne.jp/~qemu-win/TapWin32-en.html>

La aplicacion que viene como ejemplo en **Toro** requiere que se tenga instalado **OpenVPN v. 2.0.9** con el nombre del adaptador como TAP2.

Los valores correctos del protocolo **TCP-IP** del adaptador de redes son:

*IP : 192.100.200.50*

*Gateway : 192.100.200.1*

## Debug

**Toro** posee código de Debugueo que es compilado cuando se define el símbolo `{ $DEFINE DEBUG }`, en el archivo `rtl/toro.inc` se define los símbolos necesarios para iniciar los procedimientos de Debug además de otras directivas para el compilador. Parte de `toro.inc` es:

```
...
{ $DEFINE DEBUG }
{ $DEFINE DebugThreadInfo }
{ $DEFINE DebugMemory }
{ $DEFINE DebugProcess }
{ $DEFINE DebugProcessEmigrating }
{ $DEFINE DebugProcessInmigrating }
.....
```

Primero se avisa al compilador que agregue el código de debugueo, luego se define que tipo de información debe entregar el debugger. El debug se realiza a través del puerto serie COM1 , cuando el debug esta activado la salida por el puerto es como la siguiente :

```
6872263 CPU0 #0 Initialization of debugging At Time 2479036914
6899453 CPU0 #0 CPU Speed: 128 Mhz
6915628 CPU0 #0 Loading SMP system ...
7210608 CPU0 #0 CPU#0: OK, ApicID: 0
7225538 CPU0 #0 Booting CPU1 , usign local APIC
7244183 CPU1 #0 Boot Confirmation of CPU1 ---> Ok
14926773 CPU0 #0 CPU#1: OK, ApicID: 1
14942303 CPU0 #0 Booting CPU2 , usign local APIC
14961563 CPU2 #0 Boot Confirmation of CPU2 ---> Ok
22644155 CPU0 #0 CPU#2: OK, ApicID: 2
22667227 CPU0 #0 Memory initialization: 268435456 free bytes
22692662 CPU0 #0 Memory per CPU: 86682282 bytes
22713352 CPU0 #0 Memory Kernel: 8388608 bytes
22733137 CPU0 #0 Memory Region of CPU0 , start: 8388608
22756352 CPU0 #0 Memory Region of CPU1 , start: 95070890
22780187 CPU0 #0 Memory Region of CPU2 , start: 181753172
22805012 CPU0 #0 SlabHashInit - CPU: 0, size: 67108864
```

En la primer columna es el registro *time stamp counter* , la segunda indica la CPU, la tercera el thread actual sobre esa CPU y la cuarta la información del debug.

Esta es la única parte donde se utiliza la instrucción `"lock"`, para que no se corrompa la salida por el puerto serial si dos CPUs quieren enviar datos a la vez.