

Compiling and Testing TORO

Audience: for developers.

by Matías E. Vara

Last Modification: 27 / 02 / 2008

About this paper:

The idea of this simple paper is to talk about the steps to compile application using RTL **Toro** and the new implementation of Operating System inside a single binary file.

Also I will talk about testing the system and debug procedures implemented in **Toro**.

If you find broken links or have some comments please tell me.

Greet to my friend **KW**,

Matías E. Vara,

toro.sourceforge.net

matiasvara@yahoo.com .

Contents

Compilation of the RTL Toro inside the user application.....	4
Testing the System	6
Debug procedures	8

Compilation of the RTL Toro inside the user application

The compilation of RTL is very easy, first you need to download one of these packages:

Toro32-0.03-src: for i386 Architecture.

Toro64-0.03-src: for AMD x86-64 Architecture.

Once you unRAR the file, you will have the following directories:

/boot:

This directory contains the files for the initialization of the system and the application *build.exe* is used to create the image of disk to boot Toro.

/rtl:

This directory contains the minimal RTL and the units of the kernel.

/

At root directory, you will find the file *Compile.cmd* to compile the RTL **Toro** and the user application (in this case *toro.lpr*).

The general structure of user application should look like:

program toro;

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

```
uses Kernel in '..\rtl\Kernel.pas',  
Process in '..\rtl\Process.pas',  
Console 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  
  // Code of the user application  
end.
```

To compile the program, go to **Toro** directory and execute:
>*Compile.cmd*

If the compilation has successfully been completed, the minimal RTL, kernel units, drivers and user application have been compiled and the file *toro.exe* has been created.

The application *toro.lpr* is compiled using the previous procedure.

On 64 bits, the compilation of **Toro** can be done easily with **Lazarus**. **Lazarus** is the recommended IDE to program applications in **Pascal** Language (Lazarus is an IDE similar to **Delphi**), **Lazarus** is distributed under GPL license, the official web site is located at:

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

To start with Toro64, the recommended system is Windows x64 (Windows XP 64 bits or Windows 2003 64 bits edition), you first need to download the daily snapshot version 0.9.25 or latest of **Lazarus for Win64**, install Lazarus in default location, and then open *toro.lpi* in **Toro** directory.

In **Lazarus** go to *Run > Quick Compile*.

If compilation is successful, *Toro.exe* is generated in **Toro** directory

The tools necessary to compile RTL **Toro** for 32 bits are:

- *FreePascal Compiler de 32 bits v. 2.0.2* from:

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

- *NASM v. 0.98.39* from:

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

And for 64 bits the necessary tools are available at:

- *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

This packet include the binaries of **Lazarus** and **Freepascal**.

- *Yasm v. 0.5.0 for Win64* from:

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

This tool is only for compile Initialization files and are compiled by default.

NOTE:

In the file *Compile.cmd* are declared some environment variables:

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

It must contain the path where are located all the tools for the compilation

Testing the System

Once you have obtained the file *toro.exe* you must generate the boot image.

You have two ways :

1 – Go to **Toro** directory and run :

> *RunOnQemu.cmd*

This script creates the boot image and run **Qemu** emulator.

2- In **Lazarus** go to :

Run > Run

Toro is compiled and is execute *RunOnQemu.cmd* directly.

You can use any emulator for **i386** or **AMD x86-64** architecture, for example: **Bochs, Qemu, Vmware**, etc.

Qemu can emulate **i386** and **AMD X86-64** architecture, and supports up to 255 processors. To download **QEMU** the links are:

The file *RunOnQemu.cmd* has the commmand line for creates a virtual machine using **Qemu**.

Qemu v. 0.9.1 for Win32 and Win64

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

The package **Toro** you downloaded includes a script file *RunOnQemu.cmd* to test **Toro** using the virtual machine **Qemu**, for works correctly must be installed at *C:\qemu-0.8.2-windows* directory or edit the file *RunOnQemu.cmd* and put the correct directory.

Using a Filesystem on Qemu :

Qemu supports up to four hard drivers called :

hda y *hdb*, Master Controller, master and slave disk respectively.

hdc y *hdd*, Slave Controller, master and slave disk respectively

For add an image of hard driver you must edit *RunOnQemu.cmd* and write the setence :

- *hda image.img*

This command put the image of hard driver *image.img* to Master Controller, master disk in the emulator machine.

For test the Filesystem download the file LinuxRedHat.rar from : <http://prdownloads.sourceforge.net/toro/LinuxRedHat.rar?download> and unrar it in **Toro** directory. It has a **Linux RedHat** system with EXT2 FS.

Using Ethernet adapter in Qemu :

From **Toro 0.03** is included support for ethernet card compatible with NE2000 models.

To simulate a Virtual Net on Windows you will use the application **OpenVPN** which is distributed under GPL license, the official web site is located at <http://openvpn.net/> .

OpenVPN v. 2.0.9

<http://openvpn.net/release/openvpn-2.0.9-install.exe>

Download and Install **OpenVPN** , then you must edit the file *RunOnQemu.cmd* adding the sentence:

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

This command emulates an ethernet network card ne2000, with the MAC in *macaddr* and using the virtual Adapter parameter *ifname* (recommended value: TAP2)

For more information refer to the excellent tutorial :

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

To run **toro.lpr** correctly, it is necessary that **OpenVPN v. 2.0.9** is installed and the name of adapter must be TAP2.

The corrects values of **TCP-IP** parameters in network adapter are:

IP : 192.100.200.50

Gateway : 192.100.200.1

Debug procedures

Toro has some procedures to help trace and debug the execution of the code, it is enabled when the *Debug* symbol is defined, the file *rtl/toro.inc* defines the symbols needed to initialize the debug procedures and some directives for compiler.

Some code of *toro.inc*:

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

You must define the procedures in order to debug Toro.

The debug procedure uses the serial port COM1 to send the data, the typical output in *serial.txt* looks like:

```
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
```

The first column is the *time stamp counter* register, the second column is the CPU ID, the third column logs the actual thread and the last column traces the debug information.

Writing through the serial port is performed using some synchronization mechanism in order to avoid concurrent interlacing writing in the serial port by 2 different threads (the *"lock"* instruction is used in this case, and this is the only place where it is used so far).