Building and Running Singularity RDK 2.0

Getting Started

Building and running Singularity is quite simple. The Singularity source code includes all tools needed to build and boot Singularity. Because Singularity will run in a Virtual PC, creating a test deployment of Singularity is, hopefully, trivial.

This document describes the steps to build the Singularity source code and boot your Singularity image into Virtual PC or real hardware. The following sections describe each step.

1. Resources

The Singularity source code contains all of the sources and tools required to build and boot Singularity. However, the Singularity source code does not contain a source code editor; you must provide these tools yourself. You'll need approximately 250MB of free disk space to install and 1.5GB of free disk space to build Singularity plus all the applications and 512MB of available RAM.

Depending on your institution's network policy, you may need to get an IPSec Boundary Exception from your Network IT team for your PXE boot server. Please ask them for more information.

If you have a question about building or running Singularity, please ask on the Singularity CodePlex discussion forum at http://www.codeplex.com/singularity/Thread/List.aspx.

Visit http://research.microsoft.com/os/singularity for published research material on Singularity.

1.1. Install Debugging Tools for Windows

Singularity supports kernel mode debugging. You should install the latest version of the Debugging Tools for Windows from http://www.microsoft.com/whdc/devtools/debugging/default.mspx

1.2. Install .NET Framework Version 2.0 or higher

The Singularity RDK 2.0 build requires the .NET Framework Version 2.0 or higher (previous versions required only version 1.1). The version identifier of the .NET Framework version 2.0 is "v2.0.50727". If you're up to date with Windows Update, you should have it installed. To install it manually, see one of:

- http://www.microsoft.com/downloads/details.aspx?FamilyID=ab99342f-5d1a-413d-8319-81da479ab0d7&displaylang=en (.NET Framework Version 3.5 Service Pack 1)
- http://www.microsoft.com/downloads/details.aspx?FamilyID=0856EACB-4362-4B0D-8EDD-AAB15C5E04F5&displaylang=en (.NET Framework Version 2.0)

1.3. Create a Singularity Build Environment

In the root directory where you installed the RDK is a script `configure.cmd`. Run the script to create a desktop shortcut to launch a shell with the configuration required to build Singularity. The shell script `base\setenv.cmd` configures the build environment for Singularity. You can also execute it by hand from a command shell:

```
%ComSpec% /k base\setenv.cmd
```

Browse the Source

The Singularity depot currently consists of the following directory trees:

- `base\Applications` – Sources for Singularity user applications, programs with a user interface that run in their own process.
- `base\Boot` – Sources for the bootstrap loaders.
- `base\Build` – The build environment includes the 16-bit C++ compiler for 16-bit bootstrap loader, the 32-bit C++ compiler for 32-bit bootstrap loader and native components of the HAL, C# compiler, Bartok MSIL to x86 compiler, linkers, and Singularity specific build and deploy tools such as the network boot server, `bootd.exe`.
- `base\Contracts` – Sources for contracts on channels.
- `base\Distro` – Target for a full Singularity build. The build process will create a tree in this directory for the current configuration, and populate it with all executable code needed to boot and run Singularity.
- `base\Drivers` – Sources for the Singularity device drivers.
- `base\Interfaces` – Interface definitions needed to build Singularity.
- `base\Kernel` – Sources for the Singularity kernel including the components of the CLR base class libraries included in Singularity and unmanaged (native) code in `base\Kernel\Native`.
- `base\Libraries` – Sources for the Singularity libraries, code that gets loaded into multiple processes.
- `base\Services` – Sources for Singularity services, programs that run in their own process and provide services to other programs rather than provide a user interface.
- `base\Windows` – Sources for build and bootstrapping tools that run on Windows, including the network boot server in `base\Windows\bootd`.
- `docs` – Documents and presentations related to Singularity.

2. The Singularity Build

2.1. Running a Build

The Singularity RDK uses the MSBuild tool to define build targets. Information on MSBuild may be found at http://msdn.microsoft.com/msdnmag/issues/06/06/InsideMSBuild/default.aspx.

Start the RDK command-shell shortcut and then type:

```
msb Distro\Tiny.proj
```

'msb' will build the kernel, volume manager, and shell. The complete list of targets:

- Tiny – Kernel, volume manager, shell.
- Small – 'Tiny' plus network stack, disk drivers
- BVT – verification suite
- SPECweb99 – SPECweb99 suite and benchmarks
- World – Everything

All the build targets accept a `/t:Clean` switch that removes all the built files. For example:

```
msb /t:Clean Distro\Tiny.proj
```

2.2. The Build Process

The complete build process first builds any custom Singularity tools that run on Windows. In most cases, copies of these tools are already checked into the `base\Build` directory. They are built on developers machines to prevent inadvertent build breaks.

The build process compiles C# module interface definitions (`.csi` files) into interface load libraries (`.ILL` files) using the interface compiler (`csic.exe`). Interface load libraries (`.ILL` files) are minimal MSIL assemblies that can be referenced when compiling C# sources. At run time, the interface load libraries are replaced with implementation code complete MSIL assemblies (`.DLL` files). Interface load libraries are a deterministic mechanism for dealing with circular dependencies between MSIL assemblies.

The build process compiles the C# portions of the Singularity kernel into an MSIL assembly (`kernel.exe`). It then builds other C# sources that are linked with the kernel.

After all C# sources have compiled, the build process invokes the Bartok compiler to create native x86 object files from the MSIL assemblies. Bartok also generates native header files for use by C++ (`halclass.h`) and x86 assembler (`halclass.inc`) portions of the kernel.

After the native headers are generated, the build process compiles and links Windows tools that require those headers, such as the `singdbg.dll` extension for the Debugging Tools for Windows (`windbg.exe`).

The build process then compiles and links the 16-bit and 32-bit bootstrap loaders. The PE binary for the 32-bit bootstrap loader is stripped of headers and converted to a flat text file that is assembled as a

data blob into the 16-bit bootstrap loader. The 16-bit bootstrap loader consists of both x86 assembly and 16-bit C++.

Finally, the C++ and 32-bit C++ assembly code portions of the kernel (in `Kernel/Native`) are compiled. Object files generated by Bartok are linked with the x86 portions of the kernel and a time stamp to produce a PE binary, `system.exe`. The PE binary is converted into a network bootable image as a minidump file, `kernel.dmp`.

Singularity uses the Windows minidump format for its boot image. You can examine the minidump by typing `windbg -z Image\obj\kernel.dmp` on the command-line. If you open the call stack window in `windbg`, you will see that the image is configured to start at the function `Hal()` from `Hal\hal.cpp`. Because symbolic information is annotated in the minidump, windbg can be used to explore the initial system image and to map from memory addresses in the Singularity log to source code via the `ln` command.

Output from the build process is stored in `obj` subdirectories within each source directory. The `base\Kernel\obj` directory contains the final bootstrap loader, `Singldr`, and bootable image, `kernel.dmp`.

3. The Singularity Boot Process

When booting, the client screen will show the progress of the boot process. Boot typically takes less than 1 minute from the time the client PC is powered on.

3.1. Supported Boot Devices

3.1.1. Network Boot

The preferred method of booting Singularity is through the network, via the PXE boot process. When booting via this method, the 16-bit Boot Loader is started directly.

3.1.2. Cd-Rom, FAT Hard Disk, and USB Boot

Booting from a FAT or Joliet file system requires a 512-byte boot sector installed at the beginning of the boot record for a given device. The role of this boot sector is to locate, load, and execute the 16-bit Boot Loader.

4. Configuring and Running Singularity on Virtual PC

The build process is optimized for network-booting Singularity on Virtual PC. For information on booting Virtual PC on real hardware, consult Section 4.

4.1. Configure a Virtual PC

4.1.1. Install Virtual PC

4.1.2. Install the Loopback Network Adapter on the Host OS

WindowsXP – The Virtual PC help explains how to install the loopback network adapter under the index section "Loopback Adapter, installing on Windows XP" in the Virtual PC help. The instructions are repeated here for your convenience:

1. To perform this procedure, you must be an Administrator or a member of the Administrators group. As a security best practice, consider using **Run as** to run this command.
2. On the host operating system, click **Start**, point to **Settings,** click **Control Panel,** and then double-click **Add Hardware**.
3. In the **Add Hardware** dialog box, click **Next**.
4. When the **Is the hardware connected?** dialog box appears, click **Yes, I have already connected the hardware**, and then click **Next**.
5. In the **Installed hardware** list, click **Add a new hardware device** and then click **Next**.
6. In the **What do you want the wizard to do?** list, click **Install the hardware that I manually select from a list (Advanced)**, and then click **Next**.
7. In the **Common hardware types** list, click **Network adapters**, and then click **Next**.
8. In **Manufacturer** list, click **Microsoft**
9. In the **Network Adapter** list, click **Microsoft Loopback Adapter**, and then click **Next** twice.
10. If a message about driver signing appears, click **Continue Anyway**.
11. In the **Completing the Add Hardware Wizard** dialog box, click **Finish**.

Vista – Although similar to the above, finding the **Add Hardware Wizard** is a little different. On the host operating system, click **Start**, click **Control Panel**, click **Device Manager**, in the Device Manager dialog **right click on the computer name** and choose **Add legacy hardware**.

Continue with step 6 above.

4.1.3. Configure the Host OS Loopback Adapter

Configure the host operating system side of the loopback adapter interface by doing the following:

WindowsXP

1. Navigate to **Network Connections** on the host operating system.
    a. One way to do this is by right-clicking **My Network Places**, and then click **Properties**.
    b. Another is through **Start**, **Settings**, (**Control Panel**)
2. In the **Network Connections** window, find the loop back adapter. The loop back adapter will have an auto-generated connection name like **Local Connection 2** and a device name of **Microsoft Loopback Adapter**. Right click on this network connection, select **Rename** and enter `Loopback`.
3. Right click on it again and select **Properties**.
4. On the **General** tab of the **Properties** dialog box, in the **This connection uses the following items** list, select **Internet Protocol (TCP/IP)**, and then click **Properties**. Be careful not to change the check-boxes as you don't want to disable any of the protocols.

5. Select **Use the following IP address**, enter `10.99.99.1` for the **IP address** and `255.255.255.0` for the **Subnet mask**. Leave the **Default Gateway** and **DNS server** fields blank then click **OK**.
6. Click the **Advanced** tab of the **Properties** dialog box, then click the **Settings…** button in the **Windows Firewall** box. The **Windows Firewall** dialog box will appear. Select the **Exceptions** tab.
    a. Choose the **Add Port…** button add UDP port 67 for DHCP and UDP port 69 for TFTP. DHCP and TFTP are protocols used in the network boot of Singularity.
    b. OR, Choose **Add program…** and point it at <RDK Install Dir>\Microsoft Research\Singularity RDK\base\build\bootd.exe
7. In the **Properties** dialog box, click **Close**.

Vista

1. Navigate to **Network Connections** on the host operating system. One way to do this is through **Start**, **Control Panel**, **Network and Internet**, **Network and Sharing Center**, **Manage network connections** (in the left panel).
2. The rest of the steps are the same as above except as noted below…
3. Same.
4. On the **Networking** tab, choose **Properties** of **Internet Protocol Version 4 (TCP/IPv4)**.
5. Same
6. Returning back to the **Network and Internet** dialogue in the **Control Panel** choose **Allow a program through Windows Firewall** under the section **Windows Firewall**. The **Windows Firewall** dialog box will appear at the **Exceptions** tab.
    a. Follow 6a or 6b above.

4.1.4. Create a Virtual PC

You need to create a virtual machine to run Singularity. To do this, simply copy the file `base\boot\pxe.vmc` to `base\pxe.vmc`.

4.2. Booting Singularity on Virtual PC

To set up a virtual PC for use in working with the RDK, copy the Virtual PC configuration file from `%SINGULARITY_ROOT%\boot\pxe.vmc` to `%SINGULARITY_ROOT%` and give it a new name, e.g.:

```
cd %SINGULARITY_ROOT%
copy boot\pxe.vmc myPC.vmc
```

After building the RDK by running:

```
Msb Distros\<distroFile>
```

The Virtual PC can be started to boot from an ISO image generated during the build using:

```
boottest.cmd myPC.vmc.
```

The debugger can be started to connect to the virtual PC using:

```
dbg.cmd /pipe
```

In general, the debugger wants to be started before the Virtual PC to ensure the debugger so the OS can detect the debugger, otherwise it may fail to recognize the debugger later. This can be performed by combining the two commands online the command-line:

```
dbg.cmd /pipe & boottest.cmd myPC.vmc
```

The Virtual PC can be made to network boot by adding a /net argument to the command arguments:

```
boottest.cmd /net myPC.vmc
```

but network booting is significantly slower than booting from an ISO image because the network boot loader TFTPs each file in the distribution.

4.3. Booting on Virtual PC under Vista

Vista seems to honor domain IPSec for the loopback adapter, whereas XP didn't. Thus, to boot Singularity on VirtualPC under Vista, the machine must have the IPSec boundary exception (see Section 5.2).

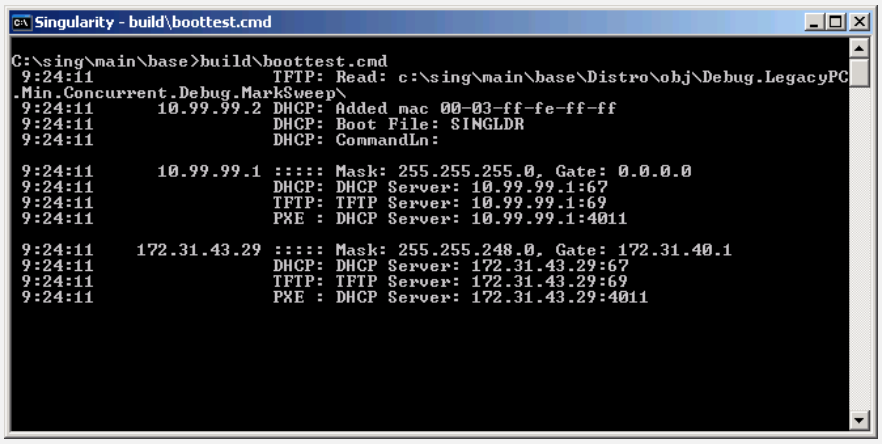
Then follow steps above.

4.4. Bootd

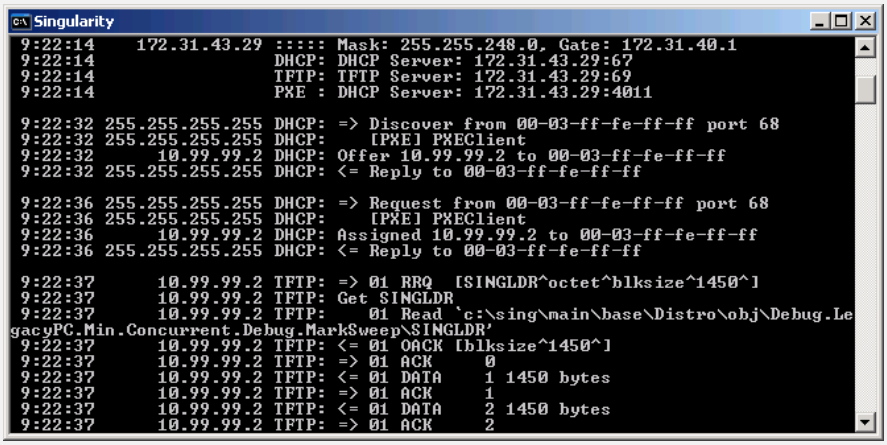
Singularity boots over a network using the PXE family of protocols. The DHCP protocol is used to get a network address. The PXE extensions to DHCP are use to determine the correct image to load. The TFTP protocol is used to retrieve the OS image. The Singularity build environment includes a simple network boot server which provides all of these protocols. The DHCP protocol can be disabled when run on a network that already provides DHCP service. Note that in the case where the boot server communicates through a loopback network to a Virtual PC then the DHCP protocol should be enabled in `bootd.exe`.

Unlike other boot servers, `bootd.exe` does not operate in promiscuous mode; it only responds to boot requests from MAC addresses listed on the command line. As such, `bootd.exe` is safe to use on shared public networks. `Bootd.exe` can also be configured either to hand out IP addresses via the DHCP protocol, as it does when booting Virtual PC clients over the Loopback Adapter, or to ignore DHCP requests so clients receive a DHCP address for another DHCP server, such as a corporate DHCP server. For more information on configuring the `bootd.exe`, type "`bootd.exe /?`" at the command prompt.

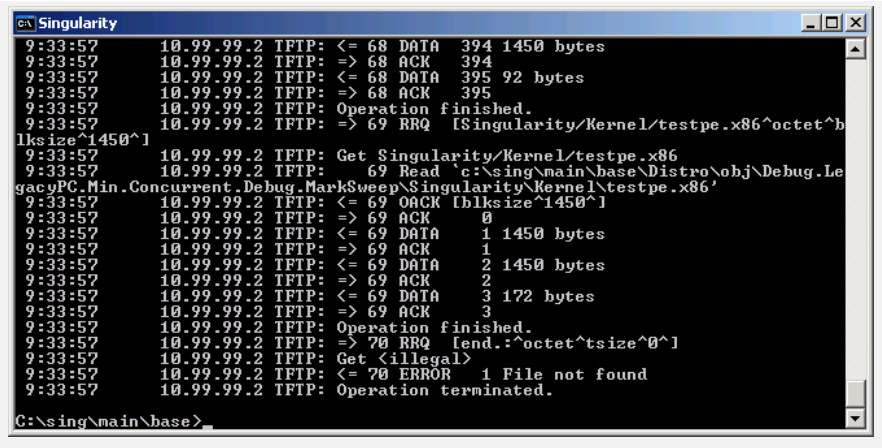
When started, `bootd.exe` displays the registered MAC addresses and bound network interfaces:



As the client boots, bootd.exe displays the DHCP/PXE requests followed by the TFTP requests:



You might notice that the last TFTP request is aborted. This is the mechanism by which the Singularity bootstrap loader indicates that it has completed downloading files from `bootd.exe`. When given the `/tftp /e` option, `bootd.exe` uses this abort to determine that a client has been booted successfully and exits. If the /e option is not provided, `bootd.exe` will continue to serve boot requests until terminated with a Ctrl-C.



5. Configuring and Running Singularity on a PC

5.1. Minimum System Requirements

If you want to boot Singularity onto a physical PC, you need a PC with at least 512MB of RAM and a Pentium II or later processor. If you want to pursue this, please contact singrdkq@microsoft.com for more information.

5.2. Network Boot

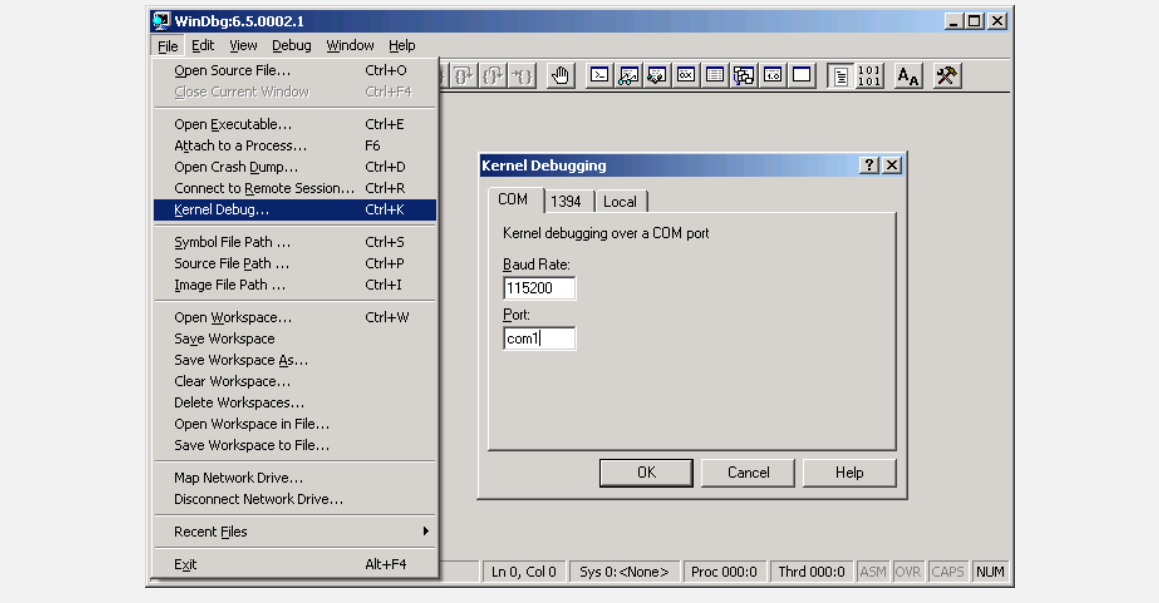
The PC needs a PXE enabled network card with the PXE enabled in the BIOS boot configuration. You'll also need to record the MAC address of the PC. Typically the MAC address is displayed as a 12 hex digit (6 byte) number during the PXE boot. Alternatively, the MAC address can be detected by looking PXE requests on the output of the `bootd.exe`.

In addition to PC hardware, you will need complete network access between your boot server (running `bootd.exe`) and your client PC. If your boot server is part of a Windows security domain, your server's IPSec policy will need to be changed or it will refuse to communicate with your client PC on UDP port 4011 which is required for the PXE boot.

The `DetectIpsecPolicy.cmd` command in `base\build` will display the current IPSec policy for a machine. To act as a boot server, the IPSec policy should be "SecNet Request Security" and not "SecNet Require Security".

Once your network is configured properly, copy the file `build\boothw.cmd` and open the new copy in an editor. The option `/m:XX-XX-XX-XX-XX-XX` describes the MAC address of your machine. Set the MAC address to the value you recorded earlier.

At this point, start the kernel debugger and change its configuration to use a direct serial cable instead of a named pipe:



Then execute your copy of `boothw.cmd`, and boot the client PC.

Firewire debug requires passing a boot option that can only be done for network boot (`boothw.cmd /dhcp /c:DBG=0`) and then invoking `boottest.cmd /1394`.

5.3. Cd-Rom Boot

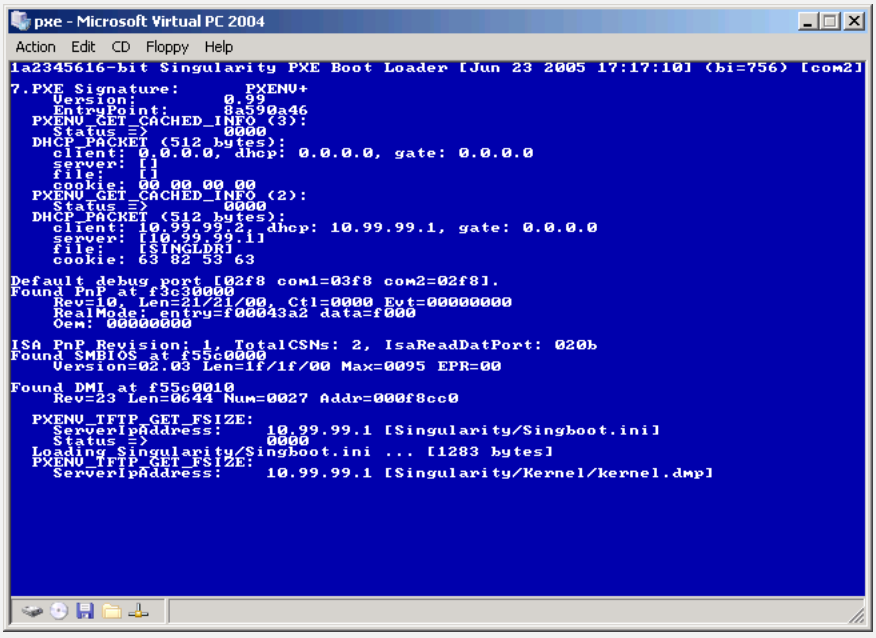
The build process creates an ISO image of the built OS. MSBuild emits the name of the ISO in the last stages of a successful build. The line starts with the string "CD-ROM image:".

Using an ISO recording tool, such as `cdburn.exe`, you may now burn this image to a disc and use it to boot your PC. Alternatively, you may capture this image directly in Virtual PC to boot from it instead of a network device.

6. Stages of the Boot Process

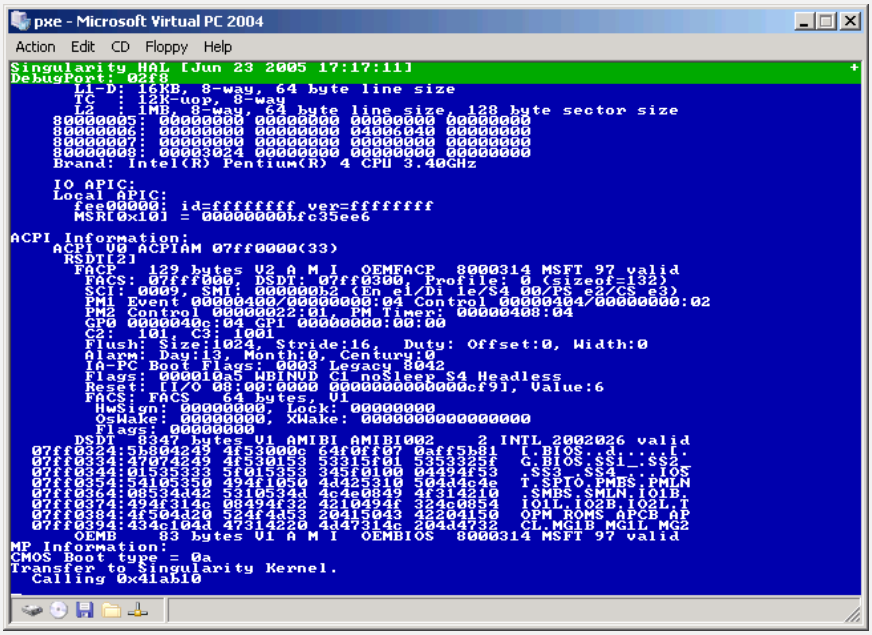
The Singularity boot goes through five distinct phases:

1. **16-bit Boot Loader** (`singldr.cpp` and `singldr0.asm` in `Singldr`). The 16-bit boot loader is loaded into the target machine by PXE or a boot sector. The 16-bit boot loader first requests the boot sequence file (`Singularity\Singboot.ini`), and then parses it to determine the filenames comprising the remainder of the boot image. The first file must be the Singularity minidump image (`Kernel.dmp`). After the 16-bit boot loader successfully loads all files listed in `Singboot.ini`, it gathers configuration information into the `bootinfo` block, disables interrupts, enables the MMU, and transfers control and a pointer to the `bootinfo` block to the 32-bit boot loader. On the x86-PC architecture, the 16-bit boot loader is the only code to interact with BIOS and the only code to allocate memory from the low 1MB of the address space.

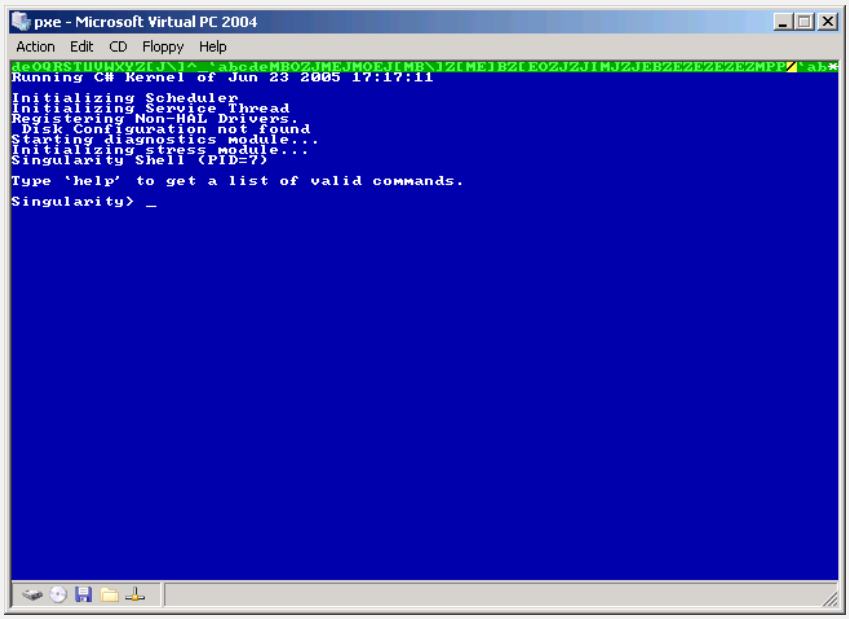


2. **32-bit Boot Loader** (`undump.cpp` in `pxe.com`). The 32-bit boot loader copies the various components of the minidump image to the virtual addresses specified within the image. The 32-

bit boot loader then passes control to the first thread listed in the minidump. By convention, the first minidump thread points to the HAL entry point.



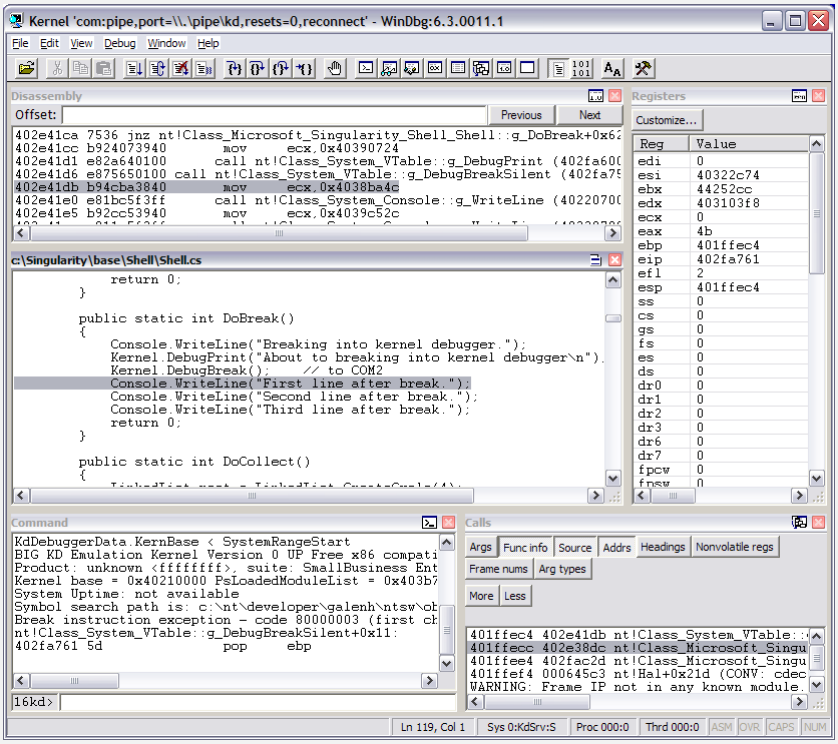
3. **Hardware Adaptation Layer (HAL)** (`Kernel\Native\hal.cpp`). The HAL initializes a base interrupt vector for early debugging and transfers control to the Kernel at `Kernel.Main` in `Kernel.cs`.
4. **The Singularity Kernel** `(Kernel\...)`. The kernel initializes the managed code runtime, performs the initial inventory of hardware devices and starts the first device drivers, and then starts the shell process.
5. **The Shell** (`Applications\Shell\Shell.sg`). The Singularity shell includes a command line interpreter and a handful of statically linked test programs. You can list the programs by typing "`help`" at the prompt.



7. Debugging Singularity

7.1. Starting the Kernel Debugger

There are two kernel debuggers: KD is a kernel-mode only console based debugger and WinDbg is a graphical user-mode and kernel-mode debugger. WinDbg can be conveniently invoked by the script `dbg.cmd` in the `build` directory. The script sets up the source code, symbols, and Singularity specific debugger extensions, and then invokes WinDbg. See section 4.2 for command line syntax.



7.2. How the debugger works

The kernel debugger communicates with a stub in the Singularity HAL via a serial port protocol. The default stub is wired to communicate using `com2` at 115200 bps. In the most common case, where Singularity is run in a Virtual PC, the `com2` port on the Virtual PC is connected to a named pipe on the host OS, `\\.\pipe\kd`.

The kernel-mode debugging protocol supported by KD and WinDbg stems from the development of Windows. For Singularity, the appropriate data structures are constructed in the debugger stub (`halkd.cpp`) to make Singularity appear like NT. The Bartok compiler generates the appropriate symbols so the debugger is able to step through Sing# and C# code as if it were C or C++.

7.3. Entering the Debugger

The system maybe broken into at any time by pressing the break button on the WinDbg panel. 

Within kernel code, the debugger may be entered by calling `DebugStub.Break()`. And from user code using ,`System.Diagnostics.Debug.Break()`.

7.4. Useful Debugger Commands

The basic debugger buttons and windows should be familiar to anyone who has used an MS debugger before. Rather than cover the basic buttons and panes, we present a list of techniques that we've found useful working on Singularity.

7.4.1. Evaluate C++ expression (??)

The `??` command evaluates a C++ expression. This provides a fast way to evaluate an expression from the command prompt (`kd>`). C# expressions can be evaluated by transmogrifying them into C++ expressions (replace . with :: when describing members).

7.4.2. Print a static member variable

Static member variables do not show up when debugging sessions, but can be accessed through the command-line with the `??` command. For instance to print the value of the static variable `done` in the shell, the interaction would be:

```
kd> ?? shell!Shell::done
bool false
```

7.4.3. Print a string

The debugger is not aware of the C# string representation, but can be displayed in the debugger. Suppose a string object called `message` is used in a function. The interaction to display it is:

```
kd> ?? *message
class String
 { [13] "Hello World!" }
   +0x000 __VFN_table : 0x3d81ec5c
   +0x004 m_arrayLength    : 13
   +0x008 m_stringLength   : 12
   +0x00c m_firstChar      : 72 'H'
```

The deference operator (*) is required because the string object message is actually a pointer to an object instance.

7.4.4. Print a member of an array

Arrays are represented as objects containing an `overall_length` field and a `values` field that the array elements. The first 3 values are usually visible from the debugger windows. Further values may be viewed with `??`. Suppose a method takes an array of integers called `numbers`, then elements of the array may viewed as follows:

```
kd> ?? numbers
class Int32[] * 0x3d539094
kd> ?? *numbers
class Int32[]
   +0x004 overall_length   : 5
   +0x008 values           : [3] 1
kd> ?? numbers->values[4]
int 7
```

7.4.5. Copy the stack to clipboard

When problems arise in pieces of code that you are not familiar with it is often handy to send a stack trace to the author of the code. The call stack window has an icon next to the close window button  . Pressing this button presents the user with a list of options and among these is the option to copy the stack to clipboard. When sending stack traces it is generally helpful to have all of the options enabled (Raw args, Func info, Source, Addrs …).

Similar functionality can be achieved via the interactive prompt using `k*` commands. `kv` is equivalent to all options enabled in the WinDbg GUI.

7.4.6. Catch Exceptions in Debugger

The debugger can catch C# exceptions as they are raised. To do this, navigate from the main window through Debug->Event Filters and select “Visual C++ exception”. Change the Execution option to “enabled”. The next exception thrown will cause the debugger to be entered into.

If you need to catch an exception that is thrown during the kernel initialization process then edit `dbg.cmd` and remove the `-c "g"` arguments supplied to WinDbg. This will cause the debugger to be entered into early during the kernel initialization. An event filter or breakpoint can be added before resuming by pressing the go button (or typing 'g' at the interactive command prompt).

7.4.7. Locate source lines and files from addresses

To locate a source file or address from an instruction pointer value use the `lsa` command at the command prompt. To locate the file associated with an instruction pointer value, use the `lsc` command.

7.4.8. Step source line and step instruction at a time

When stepping through code with the debugger it is possible to step on C# instruction at a time by selecting source mode on the main toolbar  . It is possible to step a machine instruction at a time by selecting instruction mode on the main toolbar  . These commands are also available through the interactive command prompt as `l+t` and `l-t` respectively.

7.4.9. Multiprocessor commands

When the debugger is entered on multiprocessor systems, the command prompt changes to indicate the current processor number, ie:

```
0: kd>
```

The current processor may be switch using `~<cpuNumber>` , where `cpuNumber` ranges from 0 to the number of processors in the system minus one.

7.5. Singularity Debugger Extensions

The debugger framework supports extensions. The extension mechanism primarily exists to allow groups within product groups to add functionality to the debugger through plugins. Extension commands are prefixed with an exclamation character (`!`). The debugger is unaware of extension commands and they must be entered at the interactive kd prompt in WinDbg.

7.5.1. Extensions Help (!help)

The `!help` command displays a list of the available extensions with arguments.

7.5.2. Inspect object (!object)

Attempts to cast a memory location into an object and display information about the object.

7.5.3. List threads (!threads)

Lists the threads present in the system. When invoked with `-s` argument, the stack associated with each thread is also shown.

7.5.4. Switch thread (!thread)

The `!thread` command switches the debugger context to the selected thread. This allows the user to debug specific threads from within the debugger.

7.5.5. View processes (!procs)

Lists the process objects present in the system.

7.5.6. View log entries (!log)

Singularity provides a fast and lightweight logging facility for application and kernel events. A portion of memory is used as a circular buffer for logging event entries. Each entry includes the event type, the time in CPU cycles, the thread id, a descriptive tag, and a user supplied message. Entries in the log are retrieved in a LIFO manner using `!log`.

Invoking the log command with `-h` displays the output and filtering options for the log extension.

The programmer interface to the log is contained under `Kernel\Singularity\Tracing.cs`.

7.6. Miscellaneous Debugger Tips

7.6.1. The compiler optimized the code away

The Bartok compiler performs whole program optimization optimization. As a result portions of code, variables, whole methods, may be optimized away. The Bartok command line argument `/minopt` will greatly reduce the number of optimizations performed. Setting `BARTOK_FLAGS=/minopt` at the top of the relevant `Makefile` will disable optimization for a particular project component. Alternatively, setting the environment variable `BARTOK_FLAGS` will have the same effect. The component directory will require a clean build for the change to have an effect.

8. Troubleshooting

There are a few common failures modes when building and running Singularity:

- *Symptom*: Build process displays "out of memory" on a cl16.exe step.
  - *Possible cause*: Environment variables are too long for cl16.exe.
  - *Possible solution*: Shorten environment variable definitions before calling `setenv.cmd`. Common culprits are `PATH` and `INCLUDE`.
- *Symptom:* Build process fails when running sn.exe with the error message, "`Failed to generate strong name key pair. Access is denied.`"
  - *Possible cause:* User does not have permission to host keys.
  - *Possible solution:* Add user rights to access the files under:

    ```
    C:\Documents and Settings\All Users\
        Application Data\Microsoft\Crypto\RSA\MachineKeys
    ```
- *Symptom:* Build fails with the message, "`fatal error CS0014: Required file 'alink.dll' could not be found`".
  - *Possible cause:* The C# compiler requires the .NET Framework version 2.0 which is not installed.
  - *Possible solution:* Install the .NET Framework version 2.0 or higher. For more information see, http://msdn.microsoft.com/netframework/technologyinfo/-howtoget/default.aspx.
- *Symptom:* The Virtual PC displays the message, "`Reboot and Select proper Boot device`".
  - *Possible cause*: The PXE boot floppy has been removed from Virtual PC.
  - *Possible solution*: Select "Capture Floppy Disk Image…" from "Floppy" menu, then select the PXE boot floppy image `base\boot\pxeboot.vfd`.
- *Symptom*: The Virtual PC waits at the "Microsoft Windows Remote Installation Boot Floppy" screen for a really long time.
  - *Possible cause*: Virtual PC window isn't getting enough cycles for the code's CPU-bound timeout delay.
  - *Possible solution*: Make the Virtual PC the foreground window.
- *Symptom*: TFTP server is not responding when using Virtual PC.
  - *Possible cause*: Virtual PC networking setting has not been set.
  - *Possible solution*: Follow the directions above under "Create a Virtual PC."
  - *Possible cause*: The PXE server is configured with an incorrect MAC address for the Virtual PC. The default MAC address is `00-03-ff-fe-ff-ff`, but

occasionally, Virtual PC will assign a different MAC address. The MAC address is stored in the `.vmc` file.

  - *Possible solution*: Copy the MAC address for the Virtual PC, in the line "`Node: 0003FF9150D9`" on the RIS boot screen, to the `boottest.cmd` file.

- *Symptom*: PXE boot floppy displays "`No reply from a server`."
  - *Possible cause*: The boot server isn't started.
  - *Possible solution*: Type "`boottest.cmd`" in the Singularity command shell.
- *Symptom*: PXE boot floppy displays "`No reply from a server`" and boottest.cmd does not show any requests from virtual PC host MAC address.
  - *Possible cause*: The Windows firewall is blocking DHCP and TFTP ports.
  - *Possible solution*: Open firewall settings and enable ports 67 (DHCP) and 69 (TFTP).
- *Symptom*: Bootd.exe displays, "`WSAEADDRINUSE (10048) Address already in use`," error and Singularity doesn't boot because it can't find a boot server.
  - *Possible cause*: Internet connection sharing is enabled on your computer.
  - *How to check*: Type "`ipconfig /all`" and look for "`IP Routing Enabled....: Yes`" in first block of configuration settings.
  - *Possible solution*: Disable internet connection sharing in network properties.