ROGUE WAVE
SOFTWARE

Evaluating TotalView

1 Compiling Programs. Compile your programs using the –g option. For example:

gcc -g –o *my_prog my_prog.c*

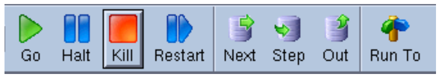
2 Starting TotalView. Enter:

totalview *my_prog* **–a** *arguments*

Or, type **totalview** from the shell to open the New Program dialog box to:

- Start a new process
- Attach to a running process
- Open a core file

3 Toolbar Buttons Defined



- **Go**: starts execution.
- **Halt**: stops execution, but you can restart from where execution stopped.
- **Kill**: kills the executing program.
- **Restart**: does a **Delete**, then a **Go**.
- **Next**: executes all code on the current line; program counter (PC) will be at the next line.
- **Step**: executes line; if the line has a subroutine, PC moves into it.
- **Out**: executes remainder of current routine; PC is on the line that called this routine.
- **Run To**: After selecting a line (click on the line, not the line number), press this button to execute all instructions from the PC until this line.

4 Setting a Breakpoint

- **Line**: click on a line number.
- **Function**: select **Action Point > At Location**, and type a function name.
- **Function**: Use the **View > Lookup Function** command, then click on the line number.
- **Search**: Use the **Edit > Find** command, then click on the line number.

5 Attaching to Already Running Programs

- Select the **File > New Program** command, select **Attach to process**, click on the program's name, then press the **OK** button.
- If you don't see the program, use the **ps** command to determine its PID (Program ID), and then select the PID within the **File > New Program** dialog box.

Always attach to a program's main thread.

6 Stopping at a Line When a Variable Equals (or Doesn't Equal) a Value

a Set a breakpoint within the loop.

b. Right click on the breakpoint icon and select **Properties**.

c. Select **Evaluate** in the dialog box.

d. Type a condition; for example:

```
if (my_variable == 0) $stop
```

7 Seeing Variable Values

- If it's a local variable, it'll be in the Stack Frame Pane. For a local or global variable, double-click on it in the Source Pane to see the value in a Variable Window, or hover your mouse over it to see the value in a tooltip.
- If it is not a complex variable (that is, it is not an array or a structure), right-click on the variable and select **Add to Expression List**.
- For arrays and structures, double-click to see all values in a Variable Window.

8 Chasing Pointer Values. If a variable's type is a pointer, double-click to see the value being pointed to.

9 Seeing Many Variables at the Same Time. You can send as many variables as you want to the Expression List window. The values in this window update every time your program stops executing.

You can also send individual structure and array elements to this window.

10 Seeing Just Some of an Array's Elements. The **Slice** area within the Variable Window lets you tell TotalView which array elements it should display. For example, typing (31:60) in Fortran or [30:59] in C or C++ restricts the display to just 30 elements.

Type a condition within the **Filter** area to restrict the display to certain values. For example, typing **> 64000** restricts the display to array elements with a value greater than 64,000.

You can combine slices and filters.

11 Graphing Arrays. Seeing array data visually is an easy way to detect outliers and patterns. Display the data graphically by selecting the **Tools > Visualize** command within a Variable Window.

12 Casting. You can change the way TotalView interprets and displays variable data by editing the **Type** field of a variable window.

For example, if you have a pointer to an array, you'll want to change the datatype from something like **int** * to **int[100]** * to see array or pointer elements.

13 Changing Variable Values

- In the Expression List and Variable Windows, click a value and edit it.
- In the Stack Frame Pane, double-click a boldface number, then edit it.

14 STL Variables. TotalView provides automatic STL type transformations to more clearly display STL data without the underlying structure. This can be toggled in the preferences as preferred.

15 Searching For Variables. Select **View > Lookup Variable** from the Process Window. The variable displays in a Variable Window.

16 Stopping Execution When a Variable's Value Changes. Use the **Tools > Create Watchpoint** command.

If the Variable Window is displaying an array or a structure, you'll need to dive on an element so that only one of the variable's elements is displayed.

17 Seeing One Element in an Array of Structures as its own Array

a Select one element.

b. Right-click and select **Dive in All**.

The window now displays an array containing just those elements.

18 Seeing a Variable's Value in Multiple Threads or Processes. From the Variable Window menu, select:

- **View > Show Across > Thread** if the program is multi-threaded, or
- **View > Show Across > Process** if the program is multi-process.

In the Stack Frame or Source Pane, right-click on the variable and select **Across Processes** or **Across Threads**.

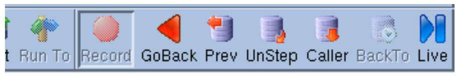
19 CLI Command Entry. Select the **Tools > Command Line** command. You can now type TotalView CLI commands within this window. Type **dhelp** for help.

20 Debugging with fork() and Execve() Programs. In most cases you must link your program with the libdbfork library that we provide. See our reference guide for more information

21 Debugging with ReplayEngine. ReplayEngine is the TotalView add-on for reverse debugging in Linux x86 and x86-64. Start it before a debugging session either from:

- the New Program Dialog Box, by selecting **Enable ReplayEngine.**
- the Process Window, by selecting menu option **Debug > Enable ReplayEngine.**

The ReplayEngine buttons on the toolbar are as follows:



- **GoBack:** Runs backwards to the nearest stop event.
- **Prev:** Moves execution in reverse, over function calls; PC moves to previous line.
- **UnStep:** Moves execution in reverse, through functions; PC moves into function calls.
- **Caller:** PC returns to the point before the function was called.
- **Back To:** When a line is selected, moves execution in reverse to the most previous execution of the line.
- **Live:** Execution and the PC are returned to current live execution location.

ROGUE WAVE
SOFTWARE

Evaluating MemoryScape

1 **Compiling Programs.** Compile your programs using the –g option. For example:
gcc -g –o *my_prog my_prog.c*

2 **Starting MemoryScape.** One way is to type

memscape *my_prog* **–a** *arguments*

on the command line.

Or, from the shell, type **memscape** to open the MemoryScape window.

- Select **Add new program**.
- Use the **Next** buttons to move through the screens to set up and start your memory debugging session.

3 **Checking for Errors.** MemoryScape stops program execution and raises an event flag before events such as the following occur:

- Freeing memory that is already freed
- Freeing the wrong address
- Freeing an interior pointer
- Misaligning blocks

Click on the event flag for details.

4 **Backtraces Defined.** When your program makes a memory request, MemoryScape records the stack frames that existed when the action occurred. This list of frames is called a *backtrace*.

5 **Showing Memory Leaks**

- Press the **Halt** button to stop program execution.
- Select the Memory Reports tab, then Leak Detection.
- Within the Leak Detection page, select either Source or Backtrace Report.

If there are leaks, MemoryScape summarizes the number of leaks and how much memory is associated with a backtrace or source line.

6 **Displaying the Heap Graphically.** Use the Heap Status Graphical Report to see how your program is using memory. Clicking on a block in the top area displays information at the bottom. Clicking on the Backtrace/ Source tab and selecting a backtrace highlights the related blocks.

7 **Filtering Information.** To reduce the amount of data displayed, you can filter information related to a process, library, source file, class name, line number, etc.

- Select **Tools > Filters...** Menu Item, or **Manage Filters** on the left.
- Select the **Add** button to create a filter. Creating a filter is similar to creating a message filter within an email program.
- After creating the filter, generate a report by clicking on the button.

8 **Tracking Memory Usage.** You can track how much memory your program is using by generating Memory Usage Reports.

9 **Block Painting.** Block painting helps you locate problems caused by accessing allocated memory before you initialize it, or accessing deallocated memory.

Block painting writes a bit pattern into newly allocated or deallocated blocks. When you see this pattern, you know that a problem is occurring.

Enable block painting by selecting **Paint Memory** within the Memory Debugging Options Page.

10 **Tracking Deallocations**

- On the Heap Status Graphical Report, right-click on a selected block and select **Properties**.
- Select **Hide Backtrace Information**, then expand the block by clicking **+**.
- At the bottom of the expanded window, select **Notify when deallocated**.

11 **Tracking Memory Blocks.** The Block Properties Window can contain information about many memory blocks. After you place a block in the window, it's often hard to identify the allocation. Adding a comment lets you remember why you're tracking a block.

12 **Comparing Memory Use.** You can use the **Export Memory Data** link on the left to write memory information to disk. At a later time, use the **Add Memory Debugging File** link on the left of the Home page to bring the information back into MemoryScape. You can examine this information in exactly the same way as normal memory information, or by using the Memory Compare page.

13 **Guarding Allocated Memory.** Guards detect when a program writes beyond the limits of your memory block. To turn them on, either select **Medium** from Basic Memory Debugging Options or select **Guard allocated memory** from Advanced Memory Debugging Options.

With guards on, MemoryScape adds a small segment of memory before and after each block that you allocate. Here are two ways to find corrupted memory blocks:

- When the program frees the memory, the guards are checked for corruption. If a corrupted guard is found, MemoryScape stops program execution and raises an event flag.
- Select **Corrupted Memory Report** from the Memory Reports page.

14 **Using Red Zones.** Red Zones allow MemoryScape to immediately notify you if your program oversteps the bounds of your allocated block. Turn them on by selecting **High** from Basic Memory Debugging Options, or by selecting **Use Red Zones to find memory access violations** from Advanced Memory Debugging Options.

With Red Zones on, a page of memory is placed either before or after your allocated block, and if your program tries to read or write in this zone, MemoryScape stops program execution and raises an event flag. Click on the event flag to see the event details.

15 **If You Have Trouble Running Your Program in MemoryScape.**

- If you're using AIX, read Chapter 4 of "Debugging Memory Problems Using MemoryScape" at **www.roguewave.com/ support/product-documentation.aspx**.
- If you're running a program that spawns processes, the problem may be that your environment isn't sending environment variables to the process. If this happens, you'll need to explicitly add libraries that we provide. See Chapter 4 of the MemoryScape book.

Using MemoryScape from TotalView

1 **Starting MemoryScape from TotalView**

- Select the **Debug > Enable Memory Debugging** command before you tell your program to start executing.

  If you don't do this, memory debugging won't work.
- Let your program run, then stop it after it allocates some memory.
- Select **Debug > Open MemoryScape**.
- Select a report, such as Leak Detection or Heap Graphical Report.

If you make changes or run your program to another breakpoint, you'll need to regenerate the report.

2 **Identifying Dangling Pointers.** When memory debugging is enabled and TotalView displays the value of a variable, it tells you if memory is allocated or if you're looking at a dangling pointer.

3 **Seeing Changes (Setting a Baseline)**

- In a Process Window, select the **Debug > Heap Baseline > Set Heap Baseline** command.
- Run your program. After stopping execution, select **Debug > Heap Baseline > Heap Change Summary** to see any memory allocations or leaks that occurred since you set the baseline.

4 **Comparing Memory Use.** If you created a baseline, go to the MemoryScape window and select your memory report. You can select the option Relative to Baseline, which shows you the information relative to the baseline you set.