

Measuring Performance on Windows Embedded Standard 7

Contents

- Summary 1
- Introduction 1
- Using the Windows Performance Analysis Toolkit 2
 - How Performance Measurement Works 2
 - Measure Runtime Performance..... 2
 - Measure State Transition Performance..... 4
- Example: Analyze CPU Usage..... 6
- Conclusion..... 9

Summary

The Windows Performance Analysis Toolkit is a powerful tool for measuring and diagnosing performance issues in Windows Embedded Standard 7. By using this tool that you can capture and view performance information for almost any part of the system. This includes your own applications.

Introduction

Performance is a very important aspect to all embedded systems. The level of performance for any given system depends heavily on the design, regardless of the operating system being used. An important part of the design process is using tools to assess performance problems and performance opportunities. Many times developers must manually instrument the operating system and programs with performance metrics, and write tools to collect and analyze these metrics. Fortunately Windows Embedded Standard 7 is built on Windows 7 and therefore takes advantage the existing instrumentation and powerful free tools that exist for Windows 7.

To improve the performance of an embedded system that is running Windows Embedded Standard 7 three primary steps are needed. The first is to determine key scenarios that are important from a performance standpoint. The second step is to measure system activities during the execution of these scenarios, and the third is to analyze the results for opportunities to improve the performance.

Starting with Windows Embedded Standard 7 there are new powerful tools that allow for better measurement and diagnosis of performance issues. The dependencies required to run them are built

into the Windows Embedded Foundation Core (every Windows Embedded Standard 7 image contains the Foundation Core) so no special steps are required to run them (with one exception which is discussed in the “Measure State Transition Performance” section).

Using the Windows Performance Analysis Toolkit

The flagship tool for measuring performance in the Windows 7 code base is called the Windows Performance Analysis (WPA) Toolkit and comes as part of the Windows 7 SDK. The WPA Toolkit enables you to measure most aspects of performance on a system. You do not have to download the complete SDK to access this tool. It can be selected by itself to minimize download time.

The Windows 7 SDK is available here:

<http://www.microsoft.com/downloads/details.aspx?FamilyID=c17ba869-9671-4330-a63e-1fd44e0e2505&displaylang=en>

The WPA Toolkit is a part of the “Developer Tools” option in the Windows 7 SDK.

How Performance Measurement Works

Throughout Windows Embedded Standard 7 events are signaled called ETW (Event Tracing for Windows) Events. These events are provided by hundreds of different providers. One of the core providers is the kernel itself. Disk read/write events, task switching events, interrupt subroutines (ISR's) entry events, Deferred Procedure Calls (DPCs) events are only a few examples. These events are collected to create event trace logs for analysis. The WPA Toolkit provides tools to capture and view these traces.

There are two main executables for capturing and viewing traces. The first is xperf.exe which is used to capture runtime traces, and to start the Graphical User Interface on traces already captured. The second is xbootmgr.exe which is similarly used to capture traces for state transitions such as restart, shutdown, hibernate, sleep, and resume, but is not used to view the traces. First we will examine runtime traces which are captured by using xperf.exe.

Measure Runtime Performance

As soon as the WPA Toolkit is installed, you can run it to capture a trace. First we will run a basic trace used for general performance investigations using xperf.exe. To execute a trace capture, type the following from an elevated command prompt:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -on DiagEasy
```

Now execute whatever scenarios you are investigating.

When you are finished executing your scenarios, stop the trace and merge all the data together into an “Event Trace Log” file by typing following from an elevated command prompt:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -d MyTrace.etl
```

The xperf.exe application will merge all the data it has collected into the MyTrace.etl file. This is a binary collection of all the trace events. Be aware that these files can become very large for longer durations.

You can now view the data by just typing:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf MyTrace.etl
```

This opens the GUI used to examine the data graphically. Figure 1 shows an example of what an event trace log file looks like in the xperfviewer tool. In this view we can see the “CPU Usage by CPU”, the “CPU Usage by Process”, and the “CPU Usage by Thread”. There are many more views under this one and you can select which views by clicking the pull out arrow on the left side of the screen.

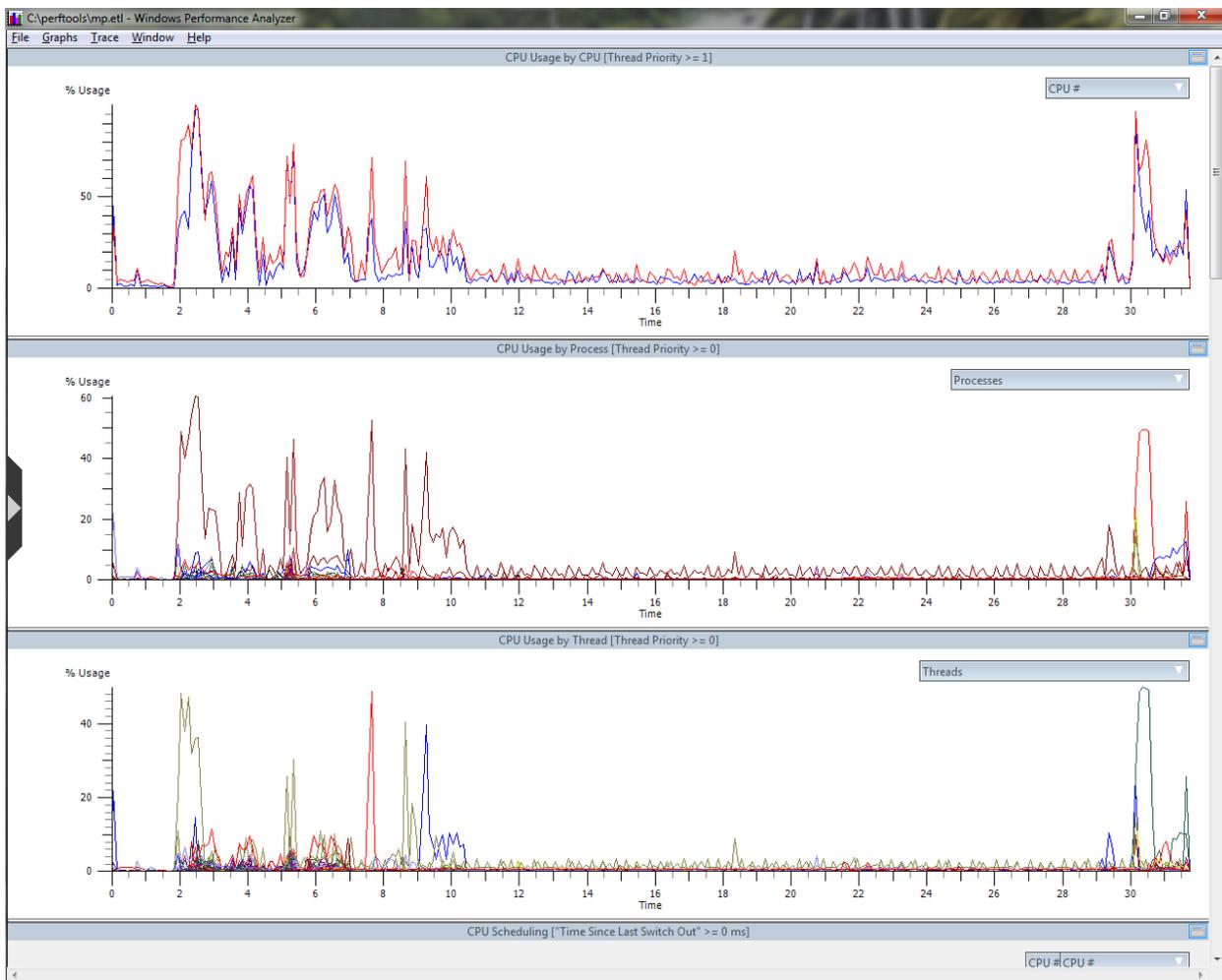


Figure 1 Viewing an Event Trace Log

In addition to viewing the graphs, you can view details about the graph by selecting a period of time (click drag to select), right-clicking on it, and selecting “Summary Table” or “Detail Graph” depending on which chart you are viewing.

In Figure 2 you can see how to select which graphs to view. Clicking the graph selection pullout midway down the left side of the screen lets you select which graphs to view and compare.

More details about how to capture runtime traces and how to use the GUI are available in the Help file provided by the WPA Toolkit.

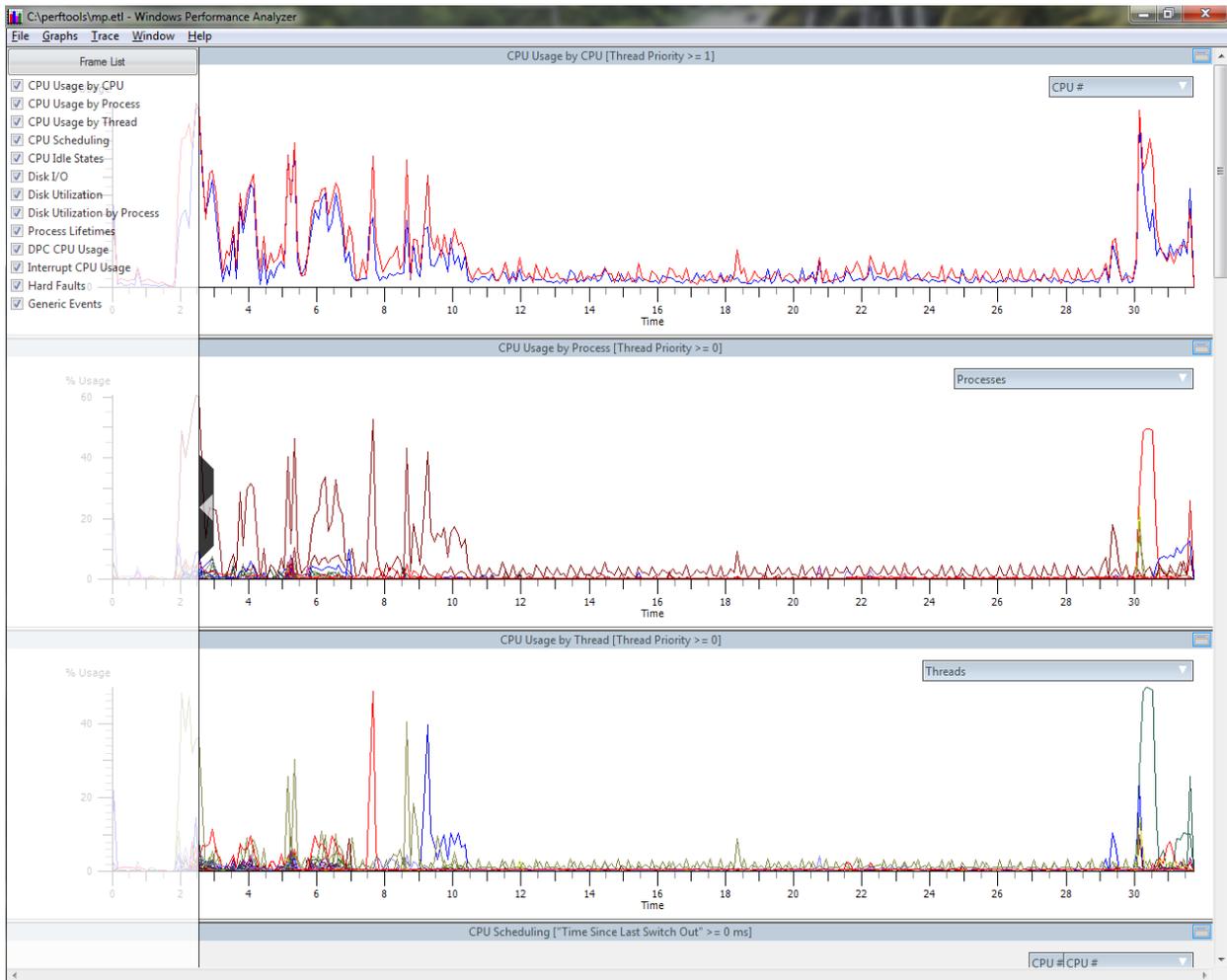


Figure 2 Selecting Which Graphs to View

Measure State Transition Performance

In addition to measuring runtime performance using xperf.exe, you can use xbootmgr.exe to measure performance metrics for boot, shutdown, suspend, hibernate, and resume.

This works much as xperf does. At the elevated command prompt, you can type the following.

```
C:\Program Files\Microsoft Windows Performance Toolkit\xbootmgr -trace boot -numRuns 3 -  
resultPath c:\myBootTraces
```

The xbootmgr.exe program will notify the user that it is shutting down the system. It will then restart three times (-numRuns 3) and collect trace information about the boot process every time. This data can then be further parsed for relevant boot data using the “boot” action. The “boot” action will parse the .etl trace file for data which is pertinent to the boot process. This includes IO times, process load times, and the boot interval times overall. It also includes several key boot metrics such as “BootDoneViaPostExplorer” which is the overall time in mS it took the system to reach a usable state plus 10 seconds. The 10 seconds are 10 accumulated seconds of idle time which are subtracted to find the point at which the user might have started an application or otherwise seen the system as useful. This is determined by measuring key system resources and establishing when the system became idle enough for the user to begin effectively using the computer. Here is how to apply the “boot” action on an .etl trace file captured by the previous command:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -i c:\myBootTraces\boot_BASE_CSWITCH_1.etl -  
o boot1.xml -a boot
```

The resulting boot.xml file will contain performance information about the boot process. Figure 3 is an example of one of these files:

```

- <results timeFormat="msec">
- <boot>
+ <processSummary numProcesses="40" numUnexpectedLonglived="30" numUnexpectedShortlived="8"
  numUnexpectedVeryShortlived="0">
- <timing bootDoneViaExplorer="18899" bootDoneViaPostBoot="36699" osLoaderDuration="1868"
  postBootRequiredIdleTime="10000" postBootDisturbance="7800" pnpBootStartStartTime="42"
  pnpBootStartEndTime="557" pnpBootStartDuration="515" pnpSystemStartStartTime="930"
  pnpSystemStartEndTime="1413" pnpSystemStartDuration="483">
+ <interval name="PreSMSS" startTime="0" endTime="1446" duration="1446">
+ <interval name="SMSSInit" startTime="1446" endTime="6672" duration="5225">
+ <interval name="WinlogonInit" startTime="6672" endTime="15831" duration="9159">
+ <interval name="ExplorerInit" startTime="15831" endTime="18899" duration="3068">
+ <interval name="PostExplorerPeriod" startTime="18899" endTime="36699" duration="17800">
+ <interval name="TraceTail" startTime="36699" endTime="143379" duration="106680">
</timing>
- <services autoStartStartTime="8530" autoStartEndTime="21471" autoStartDuration="12941">
  <serviceTransition name="PlugPlay" group="PlugPlay" transition="start"
    totalTransitionTimeDelta="436" firstCheckpointTimeDelta="327" processingTimeDelta="109"
    container="DcomLaunch svchost (548)" startedAt="8530" firstCheckpointedAt="8858"
    endedAt="8967" />
  <serviceTransition name="Power" group="PlugPlay" transition="start" totalTransitionTimeDelta="95"
    firstCheckpointTimeDelta="0" processingTimeDelta="95" container="DcomLaunch svchost (548)"
    startedAt="8968" firstCheckpointedAt="8968" endedAt="9063" />
  <serviceTransition name="DcomLaunch" group="COM Infrastructure" transition="start"
    totalTransitionTimeDelta="53" firstCheckpointTimeDelta="0" processingTimeDelta="53"
    container="DcomLaunch svchost (548)" startedAt="9072" firstCheckpointedAt="9072"
    endedAt="9125" />
  <serviceTransition name="RpcEptMapper" group="COM Infrastructure" transition="start"
    totalTransitionTimeDelta="90" firstCheckpointTimeDelta="70" processingTimeDelta="19"
    container="RPCSS svchost (616)" startedAt="9073" firstCheckpointedAt="9144" endedAt="9164" />
  <serviceTransition name="RpcSs" group="COM Infrastructure" transition="start"
    totalTransitionTimeDelta="50" firstCheckpointTimeDelta="0" processingTimeDelta="49"
    container="RPCSS svchost (616)" startedAt="9164" firstCheckpointedAt="9165" endedAt="9215" />
  <serviceTransition name="eventlog" group="Event Log" transition="start"
    totalTransitionTimeDelta="303" firstCheckpointTimeDelta="195" processingTimeDelta="107"
    container="LocalServiceNetworkRestricted svchost (696)" startedAt="9215"

```

Figure 3 Boot Action XML File

In this example the system was effectively booted in “bootDoneViaPostBoot – 10000mS” = 26.699 seconds. You can also view various other information such as which services were started, how long each service took to start, Plug n Play events, and the duration and IO activity of each interval within the boot process.

You can experiment with other actions also. To see a complete list of actions you can perform on traces type:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -help processing
```

Example: Analyze CPU Usage

One fairly widespread metric of performance is CPU usage. Every embedded system has a CPU, and every embedded system depends on that CPU to perform tasks in a timely manner. In this example I

will examine the CPU usage during the playback of an audio file in the Zune client application, as I'd like to know how much of an effect that has on my system, from a CPU usage perspective. To analyze this scenario, I must take a performance trace while playing back an audio track in zune.exe, and typing this white paper.

With Microsoft Office Word 2007 running and the Microsoft Zune Client Application playing back an audio track, I type the following in my elevated command prompt:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -on DiagEasy
```

My scenario is typing this sentence, after which I type the following into the command prompt:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -d zune.etl
```

Now I have a file that is named zune.etl which I can open by using the xperf.exe application:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf zune.etl
```

Because I am only interested in CPU related metrics, I pull out the selection bar on the left and clear everything except for "CPU Usage by CPU", "CPU Usage by Process", "CPU Usage by Thread", and "Interrupt CPU Usage". The result is in Figure 4.

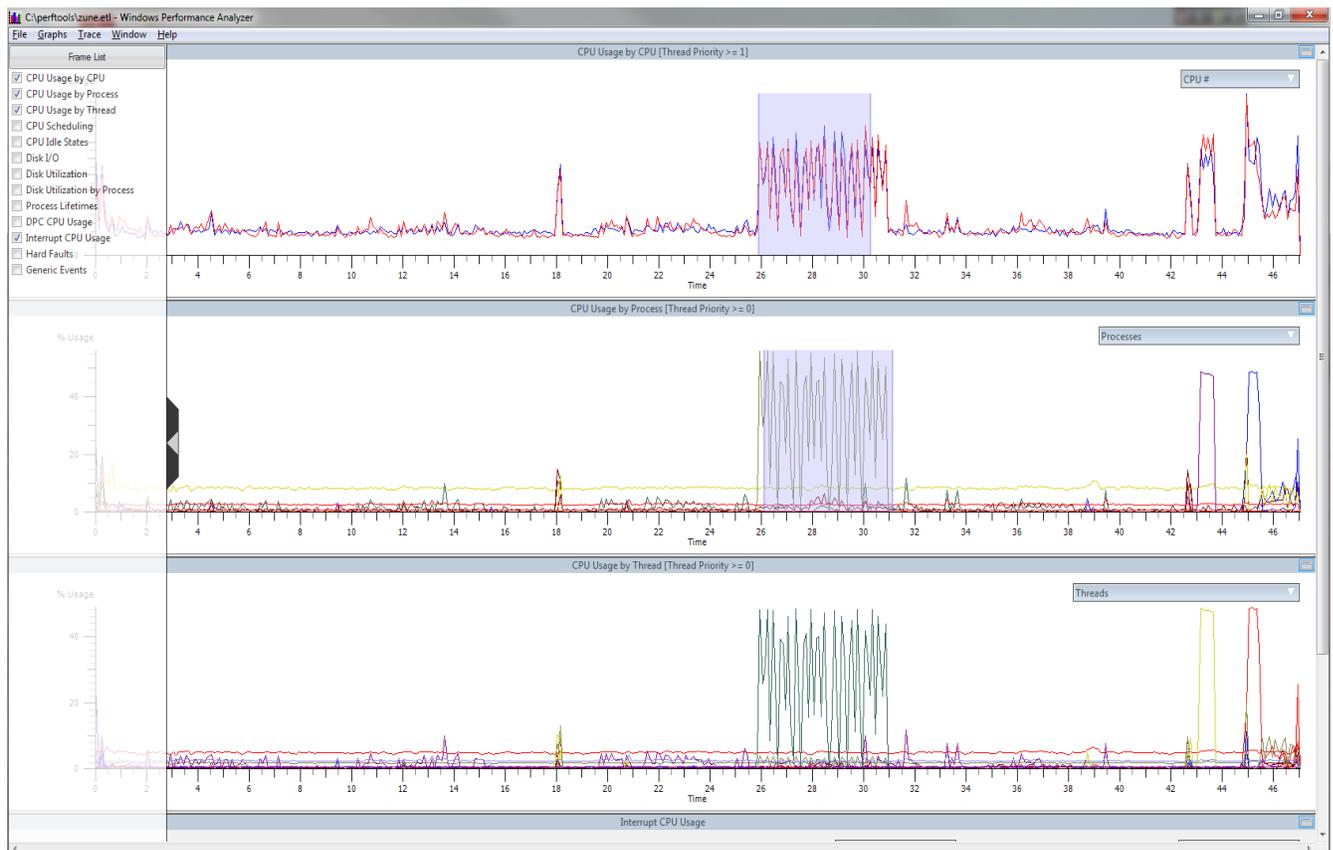


Figure 4 Example Trace View

As you can see by the highlighted section of time, the CPU appears to be very active 26 seconds into my trace. Therefore, I highlight that time period in the “CPU Usage by CPU” chart, right-click it and select “Summary Table”. A new window appears with the summary table as shown in Figure 5.

| Line | Process Name | Process | Thread ID | Cpu Usage (ms) | % Cpu Usage |
|------|----------------------|-----------------------------|-----------|----------------|-------------|
| 1 | Idle | Idle (0) | 0 | 4,609.192 847 | 52.77 |
| 2 | sqlservr.exe | sqlservr.exe (1968) | | 2,715.339 822 | 31.09 |
| 3 | Zune.exe | Zune.exe (5308) | | 738.903 094 | 8.46 |
| 4 | dwm.exe | dwm.exe (3272) | | 231.067 720 | 2.65 |
| 5 | System | System (4) | | 205.616 116 | 2.35 |
| 6 | WINWORD.EXE | WINWORD.EXE (356) | | 98.609 626 | 1.13 |
| 7 | audiodg.exe | audiodg.exe (2332) | 5,056 | 45.780 860 | 0.52 |
| 8 | csrss.exe | | | 12.910 173 | 0.15 |
| 9 | svchost.exe | | | 8.605 201 | 0.10 |
| 10 | FSysAgent.exe | FSysAgent.exe (1832) | | 4.866 106 | 0.06 |
| 11 | communicator.exe | communicator.exe (4440) | | 2.047 042 | 0.02 |
| 12 | services.exe | services.exe (540) | 2,792 | 0.800 683 | 0.01 |
| 13 | HealthService.exe | HealthService.exe (1880) | | 0.794 263 | 0.01 |
| 14 | SearchIndexer.exe | SearchIndexer.exe (3728) | | 0.497 434 | 0.01 |
| 15 | lsass.exe | lsass.exe (564) | 612 | 0.388 367 | 0.00 |
| 16 | CcmExec.exe | CcmExec.exe (2428) | | 0.342 600 | 0.00 |
| 17 | WmiPrvSE.exe | WmiPrvSE.exe (3876) | | 0.192 044 | 0.00 |
| 18 | explorer.exe | explorer.exe (3320) | | 0.177 500 | 0.00 |
| 19 | Bugger.exe | Bugger.exe (4708) | | 0.144 994 | 0.00 |
| 20 | FwcAgent.exe | FwcAgent.exe (1852) | 1,900 | 0.144 139 | 0.00 |
| 21 | mobsync.exe | mobsync.exe (5516) | 5,544 | 0.094 525 | 0.00 |
| 22 | sftlist.exe | sftlist.exe (2364) | | 0.087 682 | 0.00 |
| 23 | OfficeLiveSignIn.exe | OfficeLiveSignIn.exe (2104) | 5,924 | 0.003 421 | 0.00 |
| 24 | sftvsa.exe | sftvsa.exe (2236) | 6,072 | 0.002 994 | 0.00 |
| 25 | SRUserService.exe | SRUserService.exe (2276) | 2,336 | 0.001 711 | 0.00 |

Total CPU Usage: Non-Idle/DPC/ISR: 46.57% Idle time: 52.77% DPC/ISR time: 0.66%

Figure 5 CPU Usage Example

From the table, it is clear from line 2 that sqlservr.exe appears to be doing something taking up a significant amount of CPU during this time period. From this you can investigate why sqlservr.exe is running and possibly reduce the amount of CPU usage during this scenario. For an embedded system this may be especially important when it runs on a battery. Line three shows that during other parts of the trace, the zune.exe application is the primary user of the CPU, and takes about 8.5%, with lots of overhead in the idle task (52.8%).

In addition to viewing the trace in the UI, I can perform “actions” on the trace to process the data for key metrics. For example, running the “cpudisk” action will provide a text file summary of the CPU and disk activity for the duration of the trace. You do this by executing the following command:

```
C:\Program Files\Microsoft Windows Performance Toolkit\xperf -i zune.etl -o zunecpudisk.txt -a cpudisk
```

From this the file zunecpudisk.txt is created. Mine resembles Figure 6:

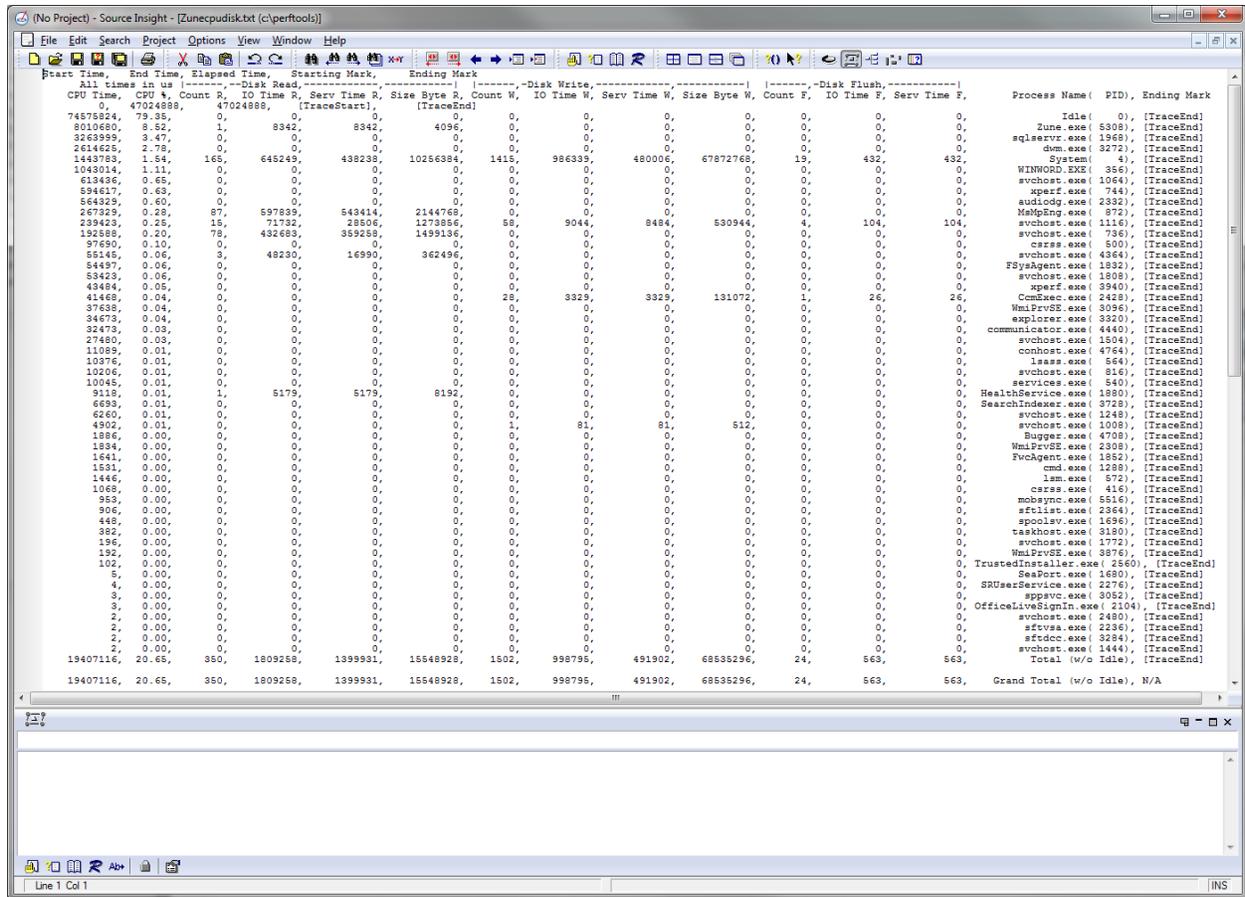


Figure 6 CPU Disk Action

You must zoom in on this to see it, but it contains average and total information for the trace on CPU and disk activity. For example, overall the zune.exe application took 8.52% of the CPU time, and 79.35% of the time was spend idle.

Conclusion

The systems and events needed to use the Windows Performance Analysis Toolkit are included in Windows Embedded Standard 7. This enables embedded systems developers to perform powerful analysis of performance on their systems. You can find the WPA Toolkit as part of the Windows 7 Client SDK [here](#).

Solutions and needs vary from one embedded project to another, but good performance stretches across all categories of embedded systems. These tools are available to aid developers in reaching this goal of achieving the best possible performance for their Windows Embedded Standard 7 device.