

Analyzing Application Performance with the Carbide.c++ Performance Investigator



Note: This information was based on an early version of Carbide.C++ from V2.0 onward this is free and you are advised to upgrade to the latest version

Introduction

The resources available to an application running on a smartphone are constrained. However, users have high expectations for the performance of applications on their smartphones, so developers are under constant pressure to create efficient, high-performance applications. Performance analysis is a valuable tool for software developers aiming to meet ever-increasing user demands for responsive, efficient applications.

In the context of software development, performance analysis is the investigation of a program's behavior using information gathered as the program executes. This type of analysis can be contrasted with static-code analysis, which investigates code before execution. The usual goal of performance analysis is to determine which parts of an application should be optimized to improve performance by reducing CPU, memory, or battery usage.

This white paper explains how to use the Performance Investigator delivered with Carbide.c++ v2.3 to analyze the performance of a third-party application. Section 2, "How performance analysis is undertaken using the Performance Investigator," shows how Carbide.c++ implements performance analysis. Section 3, "Getting started," takes the reader through the following steps: installing and configuring the Performance Investigator Profiler on an S60 3rd Edition device, running a profile session, transferring profile data to a PC, and importing the data into the Performance Investigator Analyzer. Section 4, "Analyzing performance data: a practical example," demonstrates the use of the Performance Investigator Analyzer with an example. Section 5, "Using the Analyzer," looks at other data that can be analyzed.

This paper presents the reader with the skills required to start investigating the performance of applications.

How performance analysis is undertaken using the Performance Investigator

The Carbide.c++ Performance Investigator consists of two components, as illustrated in Figure 1. The Performance Investigator Profiler runs on an S60 3rd Edition device and collects information on processes, memory, and battery usage. Then, within Carbide.c++, the Performance Investigator Analyzer allows the information collected to be analyzed.

Once the Performance Investigator Profiler has been installed on an S60 3rd Edition device, the next step is to collect data. Gathering this type of data is done by profiling — that is, by measuring the execution of a program as it runs. The Profiler is an application and a set of kernel drivers. The kernel drivers listen for function calls, and the application records the data. The result, a stream of recorded events, is called a trace. The type of data to be collected in a trace can be selected from a number of options, including the frequency and duration of function calls, memory usage, or events such as pressing buttons.

The trace data come from the kernel — not from the application being investigated. This means that the trace will include data from all processes active during the profiling session — not just the application of interest. Because of this, the Profiler marks the data to differentiate among the processes that generated the data.

It is important to note that the Profiler's method of "observation" allows for gathering data without the need to alter the implementation of the application being observed. Once a trace has been recorded, it is transferred to Carbide.c++ for analysis. Any of the usual methods for transferring data from an S60 3rd Edition device to a PC can be used. Probably the most convenient method is to use the Nokia PC Suite while the device is connected to the PC over a Universal Serial Bus (USB) or Bluetooth connection.

Once the trace data have been transferred to the PC, the Performance Investigator Analyzer is used to import and analyze the trace data. A trace can contain performance data from threads, binaries, programs, various functions, and a variety of system sources. The Analyzer then provides several methods for viewing the trace data to provide insight into the performance of the application.

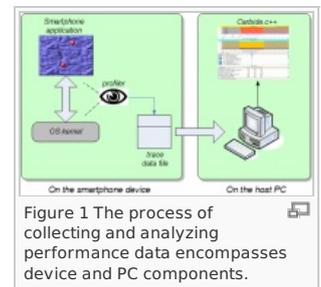
Getting started

This chapter describes the steps required to gather and make performance data available to the Performance Investigator Analyzer: installing the Profiler application, configuring the Profiler, running a Profiler session, transferring trace data to a PC, and importing the data into Carbide.c++ for analysis.

Installing the Profiler on a device

The Performance Investigator Profiler running on an S60 3rd Edition device collects performance trace data. The application collects the data in conjunction with a set of kernel drivers that allow the kernel data to be observed with no need to add special code to any application.

The Profiler must be installed on the S60 3rd Edition device on which the data will be collected. The Profiler is made available with the Carbide.c++ distribution and can be found in the C:\Program Files\Nokia\Carbide.c++ v[version number]\plugins\com.nokia.carbide.pi.-support_[version number]\pi folder. Three signed SIS files are provided — one for S60 3rd Edition devices (S60_3_0_Prof_v[version number].sisx), one for dual-processor S60 3rd Edition, Feature Pack 1 devices (S60_3_1_Prof_v[version number].sisx), and one for single-chip S60 3rd Edition, Feature Pack 1 devices (S60_3_1_Prof_v[version number]_SIZE.sisx). To confirm whether an S60 3rd Edition, Feature Pack 1 device is equipped with a single processor or dual processors, consult the Device Specifications section of the Nokia Developer website (www.forum.nokia.com/devices) and check the CPU section of a device's specification details. The Profiler is installed just as any other application for an S60 3rd Edition device would be — by transferring the application to the device or launching the installation from a PC when the device is connected via the Nokia PC Suite.



Once installed, the Profiler is available in the device's installed-applications folder, as shown in Figure 2.

Configuring the Profiler

On starting, the Profiler's main window, shown in Figure 3, displays the trace settings along with the name of the last data file into which the trace data were saved. By default, this trace file is placed on the device's memory card, but the location can be changed.

The various trace options can be set by selecting Options then Settings from the main Profiler screen. The settings are presented in three categories, as shown in Figure 4.

The Tracing options allow for specifying the types of data captured in the trace, as follows:

- Dynamic binary support — This option must be turned on to trace third-party applications installed in a device's RAM. If the application is a static executable from an image in ROM that has its own .symbol file, dynamic binary support is not needed.
- Function call capture — This option enables the recording of function calls.
- Button push capture — The Profiler can record various button events, which are used as annotations in various Analyzer graphs. This setting is useful for marking measurement points in the data collection.
- Memory usage capture — If this option is turned on, memory usage data will be recorded.
- Thread priority capture — To obtain priority data for threads, this option must be turned on.
- Power usage capture — This option enables the recording of power usage levels. The Analyzer shows the data as a global data set over the collection period, not specific to individual processes.

It should be noted that the Address/Thread tracing option shown in Figure 3 is not included in the Tracing options. This is because the Profiler always traces addresses and thread information.

Output options are shown in Figure 5. The trace can be recorded to a file or sent to a debug port. If the File system output option is selected, the trace file can be saved to a memory card or the device's C:\ drive. In addition, a file sequence number can be specified. If a "#" symbol is placed in the file name, it will be replaced with a number unique among the data files being collected. If the debug port output method is selected, special hardware must be connected to the debug port to capture the trace data. While there is less system overhead if the debug port is used, this feature is generally available only on bring-up boards used during device creation.

The Advanced options dialog is shown in Figure 6. There are two options: the Memory/Priority interval and the Power usage interval. The Profiler checks the operating system every millisecond to determine what thread is currently executing. The Memory/Priority interval tells the Profiler how often an additional check on the operating system should be made to determine the priority of the currently executing thread. The Power usage interval tells the Profiler how often to check the power usage. Both intervals are given in milliseconds, with the default as 1,000 milliseconds (or 1 second) for the Memory/Priority sampling and 250 milliseconds (0.25 second) for power usage. Note that frequent sampling may affect the device's performance and the resulting trace data, because there is significant overhead to obtaining these data.

Recording trace data

Once the options have been set, profiling can be started. This is done by selecting Options > Profiler > Start. The status of the profiling session is shown in the S60 status bar. First it will show Initializing, then change to Sampling. At this point, profiling has commenced, and any activity on the phone will generate trace data. The developer can now switch to and use the application of interest.

Profiling gathers a significant amount of data. The Address/Thread tracing option alone produces between 1 kBps and 4 kBps. It is therefore recommended that profiling sessions be limited to a few minutes' duration. Before profiling is started, it may also be useful to prepare the application of interest by opening it, creating data, and moving to a point in the application just before the start of a process whose performance is of interest.

When sufficient activities have been performed, the profiling session can be stopped. This is done by bringing the Profiler to the foreground and then selecting Options > Profiler > Stop. The Profiler's status will change to Stopping profiler and then to Finished. The trace data file will then be ready for transfer to a PC for analysis.

Transferring data to Carbide.c++ for analysis

The result of a profiling session is a trace data file. That data file is stored in the phone's file system (see settings shown in Figure 5) and needs to be transferred to a development PC running Carbide.c++ for analysis. Because the data file is a standard file, it can be transferred using any one of the usual methods for exchanging files between an S60 3rd Edition device and a PC. Use of the Nokia PC Suite with the device connected to the PC using Bluetooth technology or a USB cable is recommended. The trace file can be stored anywhere on the development PC because the Performance Investigator Analyzer can import a file from anywhere on the PC. However, it is recommended that the file be stored in the folder hierarchy of the software being developed.

Importing trace data for analysis

Once the trace file has been transferred to the PC, it can be imported into the Performance Investigator Analyzer for analysis.

Trace data can be imported by selecting Import from the File menu in Carbide.c++. The Import dialog window is then displayed, and the Performance Investigator Data option, as shown in Figure 7, should be selected.

The next dialog window, shown in Figure 8, allows the trace data file to be specified or selected through browsing



Figure 2: The Profiler is installed in the device's installed-applications folder.

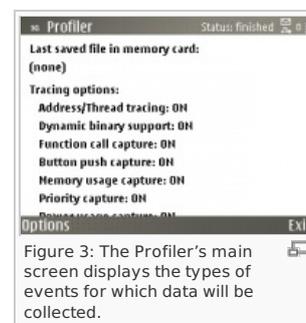


Figure 3: The Profiler's main screen displays the types of events for which data will be collected.



Figure 4: The Profiler offers three categories of options.

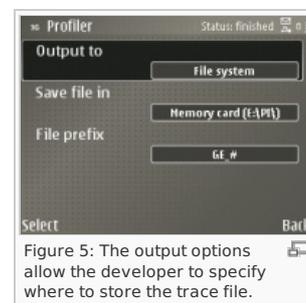


Figure 5: The output options allow the developer to specify where to store the trace file.

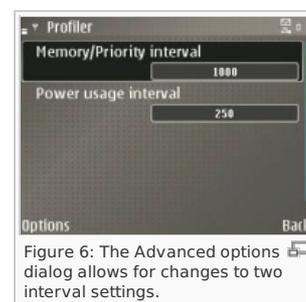


Figure 6: The Advanced options dialog allows for changes to two interval settings.

The next dialog window, shown in Figure 8, allows the trace data file to be specified or selected through browsing. This file will have a .dat extension. The file, which will be imported and converted to a format suitable for use in the Performance Investigator Analyzer, will have an .npi extension.

After the data file has been identified, the import wizard asks about the type of data to be imported. Figure 9 shows this dialog. The trace data to be imported can be one of four types: for a Carbide.c++ project; for a Carbide.c++ project on a phone where the symbol data are available; for a phone where the symbol data are available; or where there is no Carbide.c++ project and there are no symbol data available. The Profiler supplied with Carbide.c++ and installed on an S60 3rd Edition device creates files containing trace data only; the option to import ROM symbol data is not available.

After the data type of the trace file has been specified, the wizard asks, through the dialog shown in Figure 10, to which project the data file belongs. Once the project has been selected, the build configuration against which the data were collected is also selected.

Finally, the wizard asks where the imported data should be stored and what name should be given to the file, as shown in Figure 11.

Once all the import information has been entered, the developer clicks the Finish button to import the trace data. The imported data are then displayed in the Performance Investigator Analyzer.

Analyzing performance data: a practical example

This chapter examines the information available in the Performance Investigator Analyzer and explores how it can be used to identify an application's performance issues. The trace data used in this chapter are obtained using the S60 graphics example available in the S60 3rd Edition, Feature Pack 1 SDK. This example application displays two balls bouncing on the screen, as shown in Figure 12, using two different graphics methods. In one method, the ball bitmaps are displayed by copying them directly to the screen using the Window Server. This is done using the CWindowGc class to which every application has access. The second method uses an off-screen bitmap the same size as the actual display. The balls are drawn to this off-screen bitmap, then to the screen. Using this second method, the images are written to the screen in one operation, which eliminates screen flicker.

The first approach should show higher CPU usage by the Window Server. The second approach depends more on the application and should use fewer resources associated with the Window Server but place a higher load on the CPU and consume more memory because of the extra off-screen bitmap.

The example is first imported into Carbide.c++, built using the Phone Release (GCCE) [S60_3rd_FP1] build configuration, and installed on an S60 3rd Edition device. Two profiles are then taken — one for each graphics method. The resulting trace files are copied to the host PC and imported.

The button push capture option is used during profiling. This allows for identification of the time the developer displays the menu and presses the button to start the balls bouncing. This mark, which will appear as a specific point on the Analyzer data graph, makes it easier to compare the two drawing methods.

Figure 13 shows the initial Analyzer view for the first graphics method. The view shows the memory data and thread load data aligned chronologically. Moving the scroll bar for either of the graphs moves the other graph as well so that the two remain synchronized. The button presses are shown in red on the Thread Load graph. Because of the compressed timescale, the button labels are displayed on top of each other. The timescale used for the graph can be expanded by zooming in, making it easier to read the labels.

Figure 13 shows that EwSrv.exe dominates the thread load. This can be determined from the color of the load data on the graph. In addition, when the developer selects a specific time interval, the %Load and Samples columns will display information on the load associated with EwSrv.exe. Figure 14 shows an example in which a time interval of about three seconds from the start of the animation is selected. This selection is done by clicking the graph at the desired start time and dragging the mouse to the desired end time. It can be noted that the button labels are now informative and the load data are displayed. From this, it can be seen that during the first three seconds of animating the balls, the Window Server was responsible for more than 94 percent of the load, while the graphics application was responsible for less than 1 percent.

Once a time period is selected, memory statistics can also be viewed, as shown in Figure 14. The data are displayed in sections. For example, the program's static-data section is displayed separately from its heap. This information shows that the graphics application consumes approximately 116 kB of memory as it runs the animation.

The imported trace data for the second graphics method are shown in Figure 15, displaying information for a three-second period similar to that shown in Figure 14. From the new data, it can be seen that the Window Server consumes about 50 percent of the CPU, while the graphics application uses more than 40 percent. However, the memory consumption remains constant at around 116 kB.

This analysis shows that using the second graphics-handling method reduces the load on the Window Server, with the graphics application picking up the processing load. Unexpectedly, however, the amount of memory used remains the same. Examination of the application code shows that the graphics application uses the same memory resources for both methods, which explains the unchanged memory use. It should also be noted that the Profiler provides information on the power consumption of the device during the profiling session. In this case, the information is inconclusive as to whether one of the two options offers a more power-efficient implementation.

Using the Analyzer

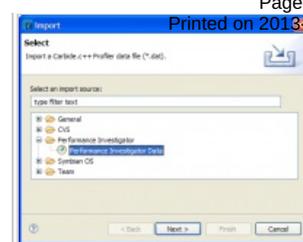


Figure 7: To import a trace file, select the Performance Investigator Data option.



Figure 8: The second step in importing a trace file is to identify the file.



Figure 9: The type of data in the trace file must be specified before the data can be imported.

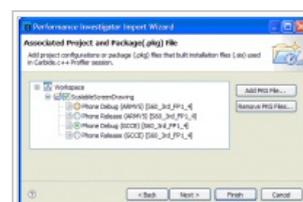


Figure 10: The importer needs to know the application's build configuration.

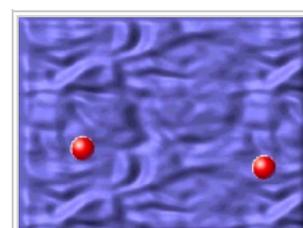


Figure 11: The final step in the import process specifies where the imported data file should be stored.

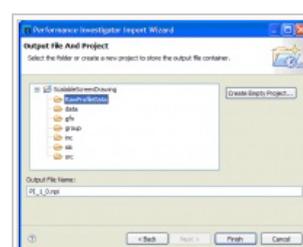


Figure 12: The S60 graphics example bounces balls around the screen using different rendering techniques.

This chapter looks at the other key features of the Performance Investigator Analyzer.

Data available for analysis

The Performance Investigator Analyzer offers separate views for analyzing threads, binaries, functions, and function calls. To switch between views, tabs on the bottom of the trace data display are used, as shown in Figure 16.

The data set for threads is the first data set displayed when data are imported.

Binaries data

Binaries data are collected and listed by binary file. If the path of the binary files for each executing process can be determined, it is displayed. Figure 17 shows a clipping of the binaries data display with the file names that were derived.

Functions data

Performance data can also be viewed by functions and details of function calls displayed, as shown in Figure 18.

The usefulness of the data depends on the nature of the application. The Analyzer derives the sources of function calls by sifting backward through the trace data, resolving the point of call for each function against the runtime address of binaries and functions offset using the map files generated by building each project. This can be a difficult process, and an application needs to use sufficient CPU cycles to show up in the trace data. If an application spends most of the time calling the Symbian Platform functions (or any library not included in the application's .pkg file), it is very difficult to find the source of a function call in a short amount of time. As a result, many calls identify the function as unknown.

Because the graphics example repeatedly calls Symbian platforms functions to draw the balls, the function source cannot be derived.

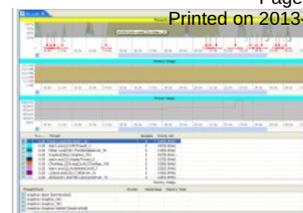


Figure 13: This data view is displayed once the trace data file has been imported.

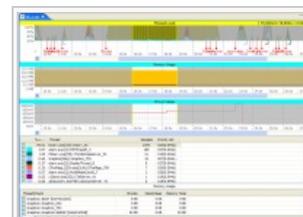


Figure 14: Zooming and selecting a time period allow for viewing data more easily.



Figure 15: Trace data for the second graphics method are shown.

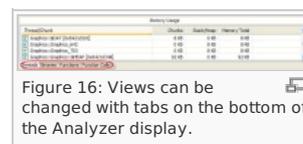


Figure 16: Views can be changed with tabs on the bottom of the Analyzer display.

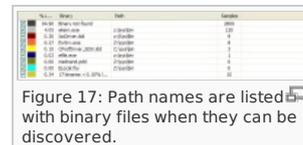


Figure 17: Path names are listed with binary files when they can be discovered.

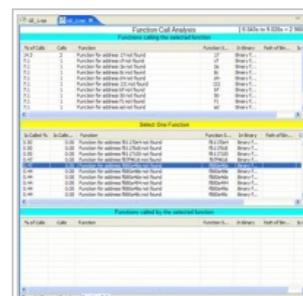


Figure 18: Function call analysis may be unable to identify calling functions in some applications.

Summary

This paper reviews how the Carbide.c++ Performance Investigator can be used to analyze the performance of an application, describing the process of collecting trace data with the Profiler application installed on an S60 3rd Edition device, transferring the data to a PC, and importing the data into the Performance Investigator Analyzer. A short tutorial shows how trace data can be used to compare the characteristics of two methods for handling graphics and to test assumptions about processor and memory usage. Finally, additional material is provided about the other information that can be obtained from the Analyzer.

The market for S60 applications is growing, and users of S60 devices have increasingly high expectations of the performance of built-in and third-party applications. Therefore, delivering optimal performance is becoming a critical aspect of application development. The Performance Investigator provides S60 developers with the tools they need to analyze an application's memory, processor, and battery performance. Using this information, developers can

more readily identify the code that is contributing to poor application performance.

The Performance Investigator is delivered as part of Carbide.c++ For more information on the range of Carbide.c++ products.

See [Latest Nokia Developer Carbide.c++ Carbide.c++ Wiki](#)



© 2010 Symbian Foundation Limited. This document is licensed under the [Creative Commons Attribution-Share Alike 2.0 license](http://creativecommons.org/licenses/by-sa/2.0/legalcode). See <http://creativecommons.org/licenses/by-sa/2.0/legalcode> for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

