Getting Down to the Bits: Xbox 360 Dump Debugging

By Cristian Ioneci
Software Development Engineer
Advanced Technology Group

Published: April 2, 2011

Abstract

Programs crash at the most inopportune times—especially during development—and, unfortunately, not only when running under the control of a debugger. Remember the bias effect studied by Murphy and colleagues? It is applicable when debugging Xbox 360 dumps. The need to determine the cause of a crash is inversely proportional to the probability of having a debugger attached when the crash occurs. If you do not have a live debugging session, the next best thing is a memory dump recorded at the time of the crash. This white paper is a primer that introduces the most common steps taken to prepare and analyze memory dump files.

Contents

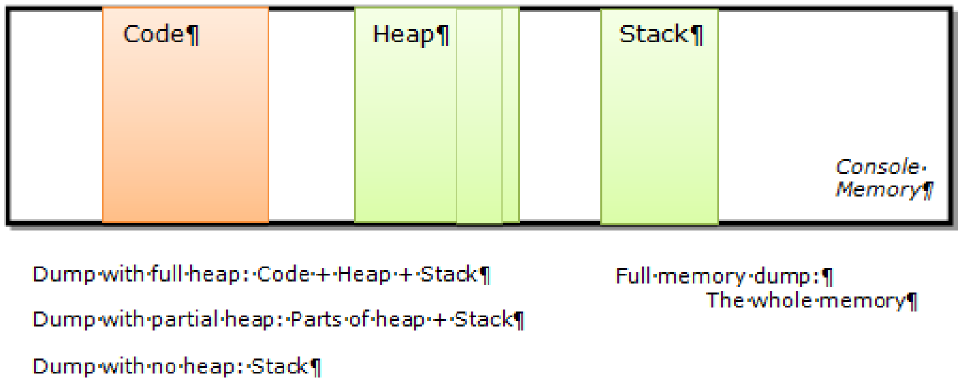
©2012 Microsoft Corporation. All rights reserved.

What's in a Dump?

A memory dump (or "dump") is a snapshot of a program's working memory. When saved with the CPU registers and other information, the snapshot enables a close reconstruction of the state of the console the moment the crash occurred. This offers a frozen view of the program execution that can be used as a starting point to infer the behavior that led to the crash.

Memory dumps often are generated by a fatal exception such as an unrecoverable memory access violation. Dumps also are programmatically triggered because an assert failed. Saving a dump, however, is not always associated with an error condition. A dump can be recorded manually or programmatically at arbitrary locations in the code, at arbitrary moments, and for arbitrary purposes—for example, to establish a base memory state for comparison with later memory configurations or to study offline and archive the threads and their memory usage.

Figure 1. Types of dumps


Code¶
Heap¶
Stack¶
Console·Memory¶
Dump·with·full·heap:·Code·+·Heap·+·Stack¶
Dump·with·partial·heap:·Parts·of·heap·+·Stack¶
Dump·with·no·heap:·Stack¶
Full·memory·dump:¶ The·whole·memory¶

The type of dump determines the amount and type of memory:

- Dumps that contain all memory in use by the program, or dumps with full heap
- Dumps that contain only the heap memory referenced by the code, or dumps with partial heap
- Dumps that contain just the stack memory used by its threads, or dumps without heap, also known as noheap dumps
- Dumps that contain the entire memory of the console

For the purpose of this paper, only these four types of dumps are addressed.

From a functional point of view, dumps with partial heap are treated by the debuggers much like a dump without heap—the binary images are needed during debugging. This paper covers the main differences between the dumps with full heap and those without heap. These two dump types can be opened with both the Visual Studio IDE debugger and two common kernel debuggers (WinDBG or kernel debugger [KD]). WinDBG, the Microsoft Windows Debugger, is a GUI-based debugger. KD is a character-based console program.

Dumps with full heap include not only the heap as the name implies, but also all memory areas (with the exception of physical memory allocations) in use by the program. Images of any

©2011 Microsoft Corporation. All rights reserved.

executable modules residing in memory at the time of the crash, together with those modules' memory areas, are included. This makes dumps with full heap the best alternative for reconstructing the state of the machine, and avoids depending on actual executable files. This comes at a cost, however, because the size of dumps with full heap can be large—tens to hundreds of megabytes (MB).

In contrast, dumps that do not save heap memory contain just the stack memory areas and context for the threads. As a result, dumps of this type are smaller—in the range of tens to hundreds of kilobytes (KB)—than dumps with full heap. For archival purposes, this makes dumps without heap memory more attractive, but may require copies of the executable binary files for the loaded modules because these dumps do not contain the executable code regions. If any binary executables are unavailable, investigation of the dump is still possible. The assembly code and the call stacks involving the given modules, however, will not display properly.

Dumps with heap memory and those without focus on gathering information about a certain user process. With an Xbox 360 console, there can be only one user process. A primary limitation of dumps with and without heap memory is that neither contains the physical memory allocated—for example, through **XPhysicalAlloc**—by the title or the system.

Save a full memory (or system) dump of the Xbox 360 Development Kit (XDK) by using the !dump command for the kernel debuggers. The saved full memory dump—an image of the entire memory of the XDK—includes the free memory and all the kernel and system areas (the whole 512 MB, or 1 gigabyte (GB) in case of an Xbox 360-GB Development Kit [XDK-GB]). A full memory dump allows the use of any command to analyze the content of the memory or state of the machine.

It can take several minutes to save a full memory dump over a network connection. A full memory dump is significantly larger than heap and noheap dumps, and could create transmission, storage, and archival problems. Because a system dump cannot be loaded and analyzed by using the Visual Studio debugger, use a kernel debugger.

Ensure you understand the trade-offs if you plan to set up a process for archiving and investigating crush dumps for your game. If you can afford the space, go with dumps with full heap because such dumps give maximum information and offer greater flexibility in processing. If, however, space is limited and you have collected many dumps of the same version of a title, it could be more advantageous to use noheap dumps and to save only one binary image.

Debugging Prerequisites

To effectively investigate a dump, it is paramount to have not only the dump file, but also these files:

- Symbol files
- Executable image files
- Source files (although not absolutely critical, these files can help immensely)

Both symbol and executable image files must be generated when the title is built, and then saved in a safe place. It is important that the configuration used to build the release version of the game has the proper settings enabled for the generation of symbol files. (There is a misconception that generating symbol files includes sensitive, symbol-related information in

©2011 Microsoft Corporation. All rights reserved.

the actual binary image). Without symbol files, the debugger displays a "sea" of numeric representations instead of the actual human-readable symbols for the addresses in memory.

Symbol Files

Symbol files can contain a public symbol table and private symbol data.

- The public symbol table contains the functions and global variables visible outside of a given compilation unit.
- The private symbol data contains additional information about all variables, structures, and types, including the line number and address pairs used to match code addresses with the source files.

There are two types of symbols files: full symbol files and stripped symbol files. The latter are referred to as public symbols, and the former are known as private symbols. The name private symbols is misleading because full symbol files contain both private symbol data and the public symbol tables. Conversely, the stripped symbols files contain only the public symbol table.

Full symbol files are more useful for debugging. Sometimes, however, you do not have access to the full symbols. This is the case with the symbols for the system components in the symbol store provided with the XDK. These are just public symbols.

The symbol files use a program database (PDB) extension. There is a signature mechanism that tries to ensure that a given PDB file matches a given executable. For this reason, ensure you save the correct symbols for the release version of your game—the release for which dumps likely will require investigation.

Executable Files

Image executable files or portable executable (PE) files are needed to properly resolve symbols and addresses when you use a dump without heap or with partial heap. While dumps with full heap do not need PE files, you should be prepared for the case when a dump without heap of your title requires investigation.

The PE file is the standard executable format for Win32 programs on a personal computer, which you could informally call Windows executables. This file is the output of the linker, before the ImageXEX tool[1] is run to generate the final Xbox 360 executable. In practice, the term PE file is a more precise; using executable might be ambiguous because it could refer to an Xbox 360 executable or to a Windows executable. An Xbox 360 Visual Studio project can generate an executable in PE (.exe) format and in Xbox 360 (.xex) format.

[1] By default, the linker automatically launches ImageXEX when linking to Xbox 360 programs. To disable the default setting, modify linker options. Once its ImageXEX automatic launch setting is disabled, the linker then writes a file with a .exe extension (in PE format). You must launch ImageXEX with the generated .exe file as an input file in order to create the final Xbox 360 executable. Remember that the conversion tool's input file is in PE format and its output file is in Xbox 360 executable format, regardless of the file extensions used.

©2011 Microsoft Corporation. All rights reserved.

If, for example, you build an Xbox 360 program from the command line, and allow the linker to automatically launch imagexex.exe, the linker writes a file with a .pe extension (in PE format), launches imagexex.exe with that as an argument, and generates a file with .exe extension (in Xbox 360 executable format).

For a command-line link used to build an Xbox 360 program, we have the .pe extension (in PE format) and the .exe (in Xbox 360 executable format). These are the default file extensions. With both Visual Studio projects and command line, file extensions can be set explicitly, overriding the defaults.

You can verify how the linker launches imagexex.exe by using **/verbose**. Extra output will be displayed by the linker, with entries similar to the following:

```
Invoking IMAGEXEX.EXE:

/nologo

/out:"MyDebug/MyXboxProgram.exe"

"MyDebug/MyXboxProgram.pe
```

The **/out:** switch sets the name and path for the output file. In the sample entry, the file uses a .exe extension. The single argument to the imagexex is the name and path to the input file (with a .pe extension in the sample).

If you do not know the format for a given executable, open the file (you can use Notepad to do this). The PE format file starts with MZ, and the Xbox 360 one starts with XEX2. Technically, MZ is the magic constant for the MS-DOS image header located before the true PE header that starts with PE.

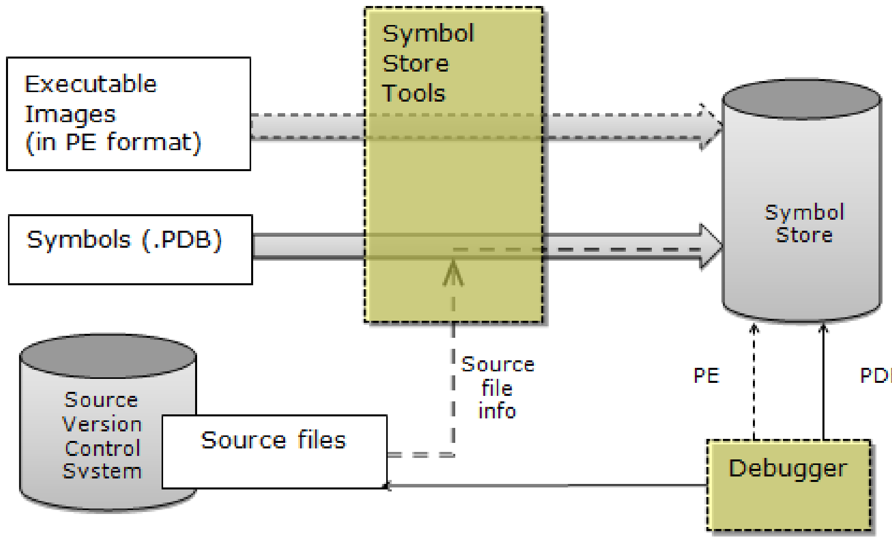
Both the symbol files and the executable image files can be saved in a symbol store, which is the recommended way to keep these files for a longer term or for archival. Symbols and images for system components are stored in a symbol store installed in %XEDK%\bin\xbox\symsrv, which is part of the XDK. With any given XDK, there is a symbol store containing symbols and images for the corresponding—and several preceding—releases of the XDK. Consider saving or archiving the symbol store for each XDK in the event you need to investigate a crash from a title built by using an earlier release of an XDK. Saving the *entire* XDK recovery/setup also provides you with access to the tools, the help, and other features.

Source Files

The symbols and executable images files are the minimum necessary to make an efficient dump investigation. An easier way to analyze a crash is to connect the assembly code to the high-level source language used to build the application. Using symbols that contain line-number information (the default setting), the debugger can offer a wealth of information such as code locations and application data in a format matching the source language. For this you need the source files used to build the application. The tools used to add the symbol files in the symbol store can be used to associate source files to the symbol files.

Source files are not copied to the symbol store. Information about the source files, however, *is* added to the store. Through the store, source files can be located in the Source Version Control System (SVCS). A visual explanation of the workflows is provided in Figure 2.

Figure 2. Symbol store usage



Thick grey arrows illustrated in Figure 2 represent actual copy operations. Executable images and .pdb files are copied to the symbol store. This is not just a dumb copy and it needs to be done by the Symbol Store tools because those tools save additional data (indexes, original file locations) for each file added to the symbol store. Symbol files also can have the source files used to create the executable associated with them. This association is done by adding some information in the actual .pdb files, after which the .pdb files containing this extra information can be added into the symbol store.

Thin black arrows signify "use" or "referenced," which means the debugger can use the identified file types.

Dashed lines means the workflow is "optional" or used "only if needed." Adding PE files to the symbol store is optional. If PE files are not needed—for example, the debugger used to analyze a dump with full heap—the debugger will not access/retrieve the PE file from the symbol store.

Information about the source is stored in the .pdb files as the files are added into a symbol store. During debugging, the debugger automatically retrieves the source associated with a given symbol file. (For more information about how to use symbol servers, see Download and Install Debugging Tools for Windows.)

Although source files are not always required, they greatly enhance the dump debugging experience. In complex cases, source files are critical in facilitating the proper understanding of the problem. For this reason, routinely backing up source files is a prudent action. And yet, it is not uncommon as the development of a title draws to a close to abandon saving the source files for each new version of the game. As indicated in the opening sentence, source files are not always needed, but their absence can add another level of difficulty when analyzing a dump and attempting to pinpoint the cause of a crash.

Support files—symbols, executable images, and source code—are essential to successful debugging, whether during a live debugging session or investigating a dump file. Care must be taken to preserve these files because it may be a long time between their creation and use for debugging.

Recording the Dump

In most cases, recording the dump is not under your control. Recording can occur automatically, or can be initiated manually and then handed to you for investigation. There are options available to control when and how a dump is saved (see Development Kit Launcher: System Settings for more information).

Involuntarily Save a Dump

Dumps can be generated involuntarily. If a program crashes, for example, the console development kit could react by saving a dump. The launcher settings that influence how to save dumps are described in the Development Kit Launcher topic.

Voluntarily Save a Dump

A dump recording operation can be triggered voluntarily by the programmer and can be generated manually or programmatically.

Manually

Voluntary dumps can be accomplished manually by using commands in a debugger connected to the console from:

- the Visual Studio debugger
- a kernel debugger by using the **!dump** command

Programmatically

Embedding certain API calls enables dumps to be saved programmatically by using:

- a **DmCrashDump** call in a program executing on the console
- a **DmCrashDump** call in a program executing on a PC as part of a debugging connection that the program running on the PC has to a program running on the console
- a **IXboxDebugTarget::WriteDump** (COM layer) call in a PC program as part of a COM debugging connection that the program running on the PC has to a program running on the console
- **DebugBreak** __debugbreak instructions

Analyzing the Dump

WinDBG is recommended for analyzing dumps that result from a lower-level crash. Debugging a crash caused by a program written in a high-level language is easier but crashes can occur in a wide-range of situations. In many situations—at the address where the system crashed (for example, in system or library components)—a high-level source code may be unavailable for

©2011 Microsoft Corporation. All rights reserved.

that area. This prompts the need to inspect, to understand, and to work with the assembly code, which can be described as a *low-level* investigation. Both WinDBG and KD are well-suited for such an investigation because of their extensive assembly code related commands and features.

WinDBG can read some dump formats that Visual Studio debugger cannot open. WinDBG has a powerful set of features, and provides a graphical user interface that mimics the Visual Studio IDE debugger. (See the WinDBG section of the Xbox 360 Kernel Debugger topic in the XDK for more information about the tool's set up, operation, and usage.)

In recent XDKs, WinDBG has powerful new functionality such as debugging of optimized code that currently is unavailable with the Visual Studio debugger. This white paper highlights the textual interface. The interface is common for WinDBG and KD and is easy to follow and archive for later processing by saving a text log of the input and output displayed in the command window.

Using WinDBG

Get started with inspecting a dump file by launching **WinDBG**. In the WinDBG command window, on the **File** menu, click **Open Crash Dump** to load the dump file. Next, you must set the paths used to search for symbols and the executable image as mentioned in the Debugging Prerequisites section.

Paths can be set by using the **Symbol File Path** and **Image File Path** options available on the **File** menu, or from a command prompt. If using the latter, use the **.sympath** and **.exepath** commands for the symbol path and executable path, respectively. Dot commands have the advantage that they can be included in debugger scripts. Both paths, together with the path to the actual dump file, can be set on the WinDBG command line, thus providing further automation opportunities. For accessing source files, similar commands (**.srcpath** and menu entries) are available.

Once the symbol and image file paths are set, use **lm** (List Modules) to verify that symbols are loaded properly in the relevant modules. An example of output may look like the following (long lines are wrapped and continuations are marked with a ➡ symbol):

```
80040000 80210000   xboxkrnld   (pdb symbols)

        ➡C:\XDK_symstore_path\xboxkrnld.pdb

817e0000 81f60200   xamd       (deferred)

82000000 82077600   prog1    C (private pdb symbols)

        ➡C:\some-path\prog1.pdb
```

The output shows that the public symbols are loaded for the kernel and match perfectly. Symbols for the Xbox Application Manager (xam), however, are not loaded yet; private symbols are loaded for the title prog1.

A letter following a module name indicates there is a problem. In the output example, the letter C in `C (private pdb symbols)`, means the checksum is zero or missing. This may or may not indicate a problem, but double check your symbols and PE files. If everything in those files looks all right, proceed with investigating the cause of the crash. A common first step is to display the stack for the current thread in the crash dump: a "k" issued in the command window or opening the Stack window from the menu.

Where to Begin

Begin your investigation by identifying what triggered the crash (that is, if it was not a planned dump recording). For an uncaught exception, this usually is announced by the debugger after opening the dump. A message similar to the following will display.

```
This dump file has an exception of interest stored in it.

The stored exception information can be accessed via .ecxr.

(f0f0f0f0.f9000000): Access violation - code c0000005 (first/second chance not
available)
```

This already gives useful information, specifying the exception code, a short description, the thread ID (f9000000). Usually the thread that caused the fault is the current thread selected in the debugger, and its execution is stopped on the instruction that triggered it. Use an **r** command to look at the instruction. The command displays the general purpose registers and the assembly instruction where the current thread stopped.

Different Faults

In most cases, the assembly instruction is at fault or is the source of the exception. For example, the instruction might have tried to write or read from a memory location that caused an access violation. Another example is when the instruction itself is a breakpoint Trap Word Immediate [twi]). There are situations when the instruction seems to be fine and there is no reason to suspect the thread. You can follow the debugger's advice and display the "stored exception information" mentioned in the message by using an **.ecxr** (Display Exception Context Record) command. This command should switch to the thread that raised the exception. When there is no exception stored in the dump, you may need to examine all other threads to identify the source of the error.

Follow the Code

Once the thread is identified, the next step is to understand how the code was executed before that instruction. The step could be trivial, especially if there is a short function where this occurred, and the code is sequential in the immediate area before that instruction. A good correspondence between the source code and the assembly instructions is helpful, which exists, for example, if the project was built with the compiler optimizations disabled. If any of these conditions—simple (short) functions and sequential code for example—are untrue, things can become difficult quickly. For a big (complex) function with lots of branches, where many paths exist that could reach the location of the exception, it can be hard to pinpoint what branch was taken. If the code is optimized, the compiler rearranges a lot of the source before actually generating the code. Assembly instructions shown for a given source line can be very misleading.

©2011 Microsoft Corporation. All rights reserved.

A new compiler included with the XDK can generate enhanced symbols information for an optimized build, which WinDBG can use to track down the content of variables with greater precision. This can help when dealing with optimized code. In this situation, manually match the assembly code with the source file. This match requires an understanding of what the instructions do. A brief example of such matching follows.

Sample of Common Steps

As a simple example, and to provide insight into the common steps taken when investigating a dump generated as a result of a failed assert, the following source code was used to generate a dump file when the assertion triggered:

```
#define XASSERT(b,msg) { if(!(b)) { OutputDebugString(msg "\n"); \

                                  __debugbreak();} }



    // . . .

    i = rand() % 4;

    XASSERT(i == 4,"i is not 4; error?");

    // . . .
```

For a build with the options for optimizations disabled, the debugger easily points to the source line with the XASSERT constant. In order to simulate the case when the code is optimized, do not trust the source code here. Gather as much information as possible, and validate our assumption about the suspected assert just by examining the assembly code.

By loading the dump in WinDBG, it can be verified that the control stopped on a twi instruction, which we expect to be the __debugbreak compiler intrinsic in the macro expansion above. All looks well so far, and in this example there is little to confuse the investigation. In a real world situation, however, it is common to have multiple asserts in the same area, making it necessary to determine exactly which assert fired. There frequently are two or more asserts verifying the validity of input parameters toward the beginning of a function, and which assert fired for a given dump is less than obvious.

At the WinDBG command prompt, try to disassemble from an address before the current stopped instruction; for example, starting with 28 bytes before:

```
0:000> u .-0n28

prog1!main+0x28 [c:\path\to\prog1.cpp @ 13]:

82010028 3cc08205 lis          r6,-32251            ; 8205h

8201002c 80a60be0 lwz          r5,0BE0h(r6)

82010030 2f050004 cmpwi        cr6,r5,4
```

```
82010034 419a0014 beq          cr6,prog1!main+0x48 (82010048)

82010038 3c808200 lis          r4,-32256            ; 8200h

8201003c 38640e50 addi         r3,r4,3664           ; 0E50h

82010040 48000159 bl           prog1!OutputDebugStringA (82010198)

82010044 0fe00016 twi          31,r0,22      ; 16h
```

Note the call to OutputDebugString. If there are multiple asserts with different associated messages, its argument can be used to identify which assert we are looking at. Another source of information is the comparison instruction (cmpwi) performed earlier and the test instruction (beq). These two instructions also can be used to match the behavior seen in the source file, and to validate the initial hypothesis that the correct assert is being scrutinized.

In the case of an access violation, you may not have much associated information and easily recognizable source code constructs, and you will have to track down the life of the values involved and figure out how they ended up in a bad state.

Conclusion

Preparing the correct support files is half the battle when it comes to investigating dump files. Be familiar with the pros and cons of each type of dump and plan accordingly. A good understanding of PowerPC assembly language also is essential in investigating lower-level issues and situations where code optimizations make it difficult to establish a correspondence with the source code.

To be successful at dump debugging requires practice. Just remember to leave the minute details to the machine!

References

Debugging Tools for Windows Package

Xbox 360 Kernel Debugger

©2011 Microsoft Corporation. All rights reserved.