|  |
| --- |
| Mago Debugger Inner Workings |
| Debugging Technology |
| Aldo Núñez |

# Introduction

Mago Debugger was made without regard to what user interface it would run with. Although it is packaged in a Visual Studio plug-in, different parts of Mago were made so they could be reused in different programs and independently of each other.

* **Execution Agent** controls a debuggee and abstracts breakpoints and stepping.
* **Expression Evaluator** determines the resulting value of D expressions.
* **Symbol Reader** handles the debug information of a program.
* **Debug Engine** ties the other parts together in a package that Visual Studio interacts with.

This breakdown allows building tools that use only certain features. For example, a trace logger for any programming language can use the Execution Agent. The breakdown also helps in testing.

# Windows Debugging

The operating system provides the framework for running, controlling, and inspecting a process. With cooperation from the hardware, the Windows API offers the necessary elements for debugging a process.

The Windows API allows control of high-level elements that the operating system already manages: processes, threads, and memory. It also allows detecting when certain events happen in a process.

Low-level details, such as what breakpoints look like and how to single step, are not abstracted. In such cases, an understanding of the facilities offered by hardware is necessary. Even so, Windows enables the needed interaction with the hardware.

## Running Processes

The normal course of a program when it runs is to

1. Start a process
2. Run CPU instructions
3. End

On some error conditions a Structured Exception is thrown which is either caught and handled by the process itself, or is unhandled which ends the process.

## Debugging a Process

The goal of debugging is to inspect the state of a process at a moment in time, and to control the execution of the process.

If we were to let a process run its normal course as described above, we wouldn’t be able control the process’s execution because the process is already controlling its own execution. We also wouldn’t be able to inspect the state of the process at a given moment, because the state is always changing.

### Break Events

In order to take control, some event must happen in the debuggee that breaks the flow of control of the process. Windows defines certain conditions that when triggered and a debugger is attached, make the OS suspend the debuggee and give control to the debugger.

Process start

Process end

Thread start

Thread end

Module (DLL) load

Module (DLL) unload

OutputDebugString

Exception

When one of these events happens, Windows stops execution of the process at the exact spot in the thread that triggered the event, and it suspends all other threads. At this point the debugger can freely inspect the state of the process and set up the process to run as it wants. The debuggee will not run again until explicitly told to, by way of the ContinueDebugEvent API.

### Debuggee Modes

In this way, a debuggee can be defined to be in one of two states at any given moment:

Break mode – the debuggee is suspended as a result of a break event, and the debugger is in control.

Run mode – the debuggee is allowed to run freely.

A debuggee will enter break mode regardless of whether its debugger has noticed this change of state. On a break event, Windows will automatically add a notification of the event to a queue associated with the debuggee. The debugger can pick up this notification by calling WaitForDebugEvent. The event can then be processed, and we can then tell the debuggee to run again by calling ContinueDebugEvent. The debuggee will not run again until this call.

StateChange.emf

Figure A debuggee's break and run mode cycle.

The debugger can debug more than one process at a time. Debug events from each debuggee are added to the same queue. ContinueDebugEvent takes the process ID of the debuggee to resume, so debuggees can be resumed in an order different from the break events that suspended them.

### Debugger Threading

We can start debugging a process in two ways:

* CreateProcess with the flag DEBUG\_PROCESS or DEBUG\_ONLY\_THIS\_PROCESS – a program starts running, and the caller is its debugger
* DebugActiveProcess –attaches to a program that’s already running

After calling these APIs, the calling thread is linked to the debuggee and becomes the debugger for that process. That debuggee can then no longer be debugged by any other thread in any process until the thread detaches. Before detaching, calls to Debug APIs related to the debuggee will fail on any other thread.

Also, there are operations that are best handled in break mode, or during the time between dispatching of debug events. In this case, we can enforce those conditions by running the operations on the debugger thread. Some such operations are writing memory and setting a breakpoint.

This imposes constraints on how threads are organized in the debugger application. A further constraint is the fact that WaitForDebugEvent is used to poll for event notifications. There is no other notification mechanism. One way to organize threads is to have a dedicated thread for polling debug event notifications; and marshalling commands that affect the debuggee from another thread (like the UI thread) to the debugger thread.

BasicThreading.emf

Figure Commands are dispatched to the monitor thread which takes turns processing them and events that are queued from debuggees.

We alternate between processing events and processing commands. This keeps both tasks responsive. Because the amount of events can greatly overtake the amount commands, we give preference to event processing by using a small timeout for event waits and 0 timeout for command waits.

## Operations

The Windows Debug APIs offer limited set of operations for manipulating and inspecting debuggees. But they are powerful enough to use as the basis for more sophisticated operations.

### Inspecting

Use these functions to access any block of memory in a debuggee:

ReadProcessMemory

WriteProcessMemory

The registers of a thread can be accessed by calling:

GetThreadContext

SetThreadContext

### Controlling

While stepping, it’s useful to have fine grained control over what threads run:

SuspendThread

ResumeThread

### Exceptions

In the discussion of break events, breakpoints, stepping, and forceful breaks were not mentioned. On Windows, Exception notifications are used for these other notifications, in addition to arbitrary system and user exceptions.

Some predefined exceptions are:

EXCEPTION\_ACCESS\_VIOLATION

EXCEPTION\_BREAKPOINT

EXCEPTION\_ILLEGAL\_INSTRUCTION

EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

EXCEPTION\_SINGLE\_STEP

EXCEPTION\_STACK\_OVERFLOW

Of these, two are used in detecting breakpoints, stepping, and forceful (async) breaks on the x86 architecture: EXCEPTION\_BREAKPOINT and EXCEPTION\_SINGLE\_STEP.

## Implementation Notes

Before linking a thread with a debuggee, calls to WaitForDebugEvent will not wait the timeout amount and will fail with ERROR\_HANDLE. This is an expected failure that a debugger should work around.

The Debug APIs that have affinity to a thread are:

WaitForDebugEvent

ContinueDebugEvent

DebugActiveProcess

DebugActiveProcessStop

TerminateProcess will not destroy a process immediately, if it’s in break mode. The process will completely shutdown only once you flush the event queue, or call DebugActiveProcessStop to detach immediately. To flush the event queue, read and continue from every event from the debuggee until the Process Exit event is handled.

If a process is started suspended (using CREATE\_SUSPENDED), then no events are fired until the main thread is resumed. Calling TerminateProcess also allows pending events to be fired, including the Process Exit event caused by the shutdown request.

Even though ReadProcessMemory and WriteProcessMemory say the return a value indicating how many bytes were read or written, the value is never set.

In a debugger, memory protection doesn’t seem to affect the memory access functions when referencing debuggee memory, except for PAGE\_NOACCESS and uncommitted memory.

# Instructions and Threads

Since no debugging can take place without being able to step and break reliably, the mechanisms for performing these two operations were the first to be made. They make up the foundation of all other systems in the debugger.

## Process Startup

At the beginning, a debuggee reports process start and module load events. Following this, there’s an EXCEPTION\_BREAKPOINT in a system DLL (ntdll.dll). Before this point, the debuggee is not ready for running. This exception should be marked as handled when calling ContinueDebugEvent.

## Process Attach

The same notifications from process startup will also be sent for process attach, even though the process was already running. This provides consistency and a definite way to hook onto the debuggee.

## Breakpoints and Stepping

Setting up and handling breakpoints and stepping depend on the processor that the debuggee is running on. The operations for handling exceptions and reading and writing memory and registers are the only ones that Windows provides. A debugger will use these functions and its own processor-specific methods in order to decode instructions, patch breakpoints, and set stepping modes.

## Controlling an x86 Debuggee

There are two mechanisms provided by the x86 architecture that are used for breakpoints (BP) and stepping:

Software breakpoint: the one-byte “int 3” instruction (CC16)

Instruction single step: the Trap Flag (10016) and Debug Exception

These two features are enough to implement complex stepping modes.

### Instruction Single Step

Setting the Trap Flag in the Flags register will do the following:

1. On continue (resume), the single instruction at the instruction pointer in the current thread will run.
2. On that thread, an EXCEPTION\_SINGLE\_STEP will trigger.
3. The Trap Flag will be cleared automatically.

Instruction Single Step (SS) by itself is not enough to implement things like *Step Range*, but can be combined with software breakpoints to offer such abilities.

### Breakpoint Patching

The simple way to set a breakpoint at address *A* is:

1. In break mode, read and store the byte at *A*.
2. Write the “int 3” instruction at *A*.
3. Flush the instruction cache.

When removing the breakpoint, we write the original byte back where it came from.

We’ll call the act of putting a breakpoint instruction in memory “patching” the breakpoint. Removing it by putting the original byte back is called “unpatching” the breakpoint. By extension, a patched BP is one that has been written to memory.

#### Single threaded Breakpoint Handling

After hitting a BP, the situation is a little more complicated.

1. After entering break mode because of the BP, we immediately write back the original byte.
2. Change the instruction pointer to the start of the instruction – to address *A*.
3. Enable single step mode.
4. Store an indication that the breakpoint needs to be restored after the single step.

After we resume the debuggee, only the instruction we restored will run. Then the single step exception will fire, and we’ll restore the BP.

We’ll call the act of temporarily unpatching in order to get past the instruction with the BP “resuming from a BP”. Patching the BP again after a SS is “restoring the BP”.

BP Lifecycle x86.emf

Figure The lifecycle of a software breakpoint on x86.

#### Multithreaded Breakpoint Handling

Actually, those were the steps to use for a single threaded environment. In a multithreaded environment, there is a problem. We temporarily unpatched the BP, intending to run a single instruction. What that means, though, is that we end up resume all threads, and some of them might have been about to hit that BP, but are no longer able to. By mistake we’ve changed the behavior of the program, or at the very least, of our debugging session.

To safeguard the BP restoring mechanism above, we change the BP hit handling as follows:

1. After entering break mode because of the BP, we immediately write back the original byte.
2. Change the instruction pointer to the start of the instruction – to address *A*.
3. Enable single step mode.
4. Store an indication that the breakpoint needs to be restored after the single step.
5. **Suspend all other threads.**

After we resume the debuggee, only the instruction we restored will run. Then the single step exception will fire, and we’ll restore the BP **and resume the other threads**.

Multithread SS.emf

Figure Achieving a single step with a breakpoint requires suspending all other threads.

### Breakpoint Patching in the Mago Execution Controller (Exec)

Exec goes beyond the ability of simple breakpoint patching by allowing patching in run mode and providing a way to share breakpoints.

#### Setting Breakpoints in Run Mode

Normally you would set a breakpoint in break mode, because that’s when it’s safe to write to the debuggee’s memory, alter its code path, and change an instruction.

Exec suspends all threads in order to patch the breakpoint. Then it resumes all threads. So, the operation becomes safe to do in run mode, but only on the debugger thread, to not interfere with event or command handling.

#### Breakpoint Sharing

There are clients (Visual Studio in particular) that need to set and remove the same breakpoint in different contexts without one context knowing about the others. There’s also an internal client that needs this ability – stepping.

A cookie identifies a context for setting or removing a BP. Exec keeps a list of cookies for each BP. A BP is patched if and only if it has cookies.

Clients can then freely set and remove BPs without regard to whether other clients need to have the BP set.

### Complex Stepping

There are several factors that affect how a step operation is done:

Step direction: In, Over, Out

Step unit: Instruction, Line

Different instructions will require stepping in different ways. For example, a SS is all that’s need to step over a mov instruction; whereas we need to run to a BP that we set at the instruction after a call in order to step over it.

Additionally, these stepping operations might have to interact with resuming from user BPs. Remember that resuming from a BP involves unpatching the BP and single stepping the original instruction. We have to determine what instructions can safely be single stepped to resume from a BP and which can’t. The idea is that: if we can single step and land outside of the target instruction, then we can safely resume from the BP with a SS. Otherwise, the SS is not safe, and we need another way to resume from the BP.

These conditions are used to define three classes of instructions.

|  |  |  |  |
| --- | --- | --- | --- |
| Class | Example instructions | SS step over | Safe resume with SS |
| RepString | rep mov | No | No |
| Call | call | No | Yes |
| Other | mov, add, jmp | Yes | Yes |

As seen from the table the RepString class doesn’t support safe resuming with an SS; after single stepping, the instruction might stay on the same instruction.

With the goal of determining what class an instruction belongs to, Mago implements some decoding routines that return the classes of instructions passed to them.

#### Steppers

Stepper objects are used to encapsulate the algorithm for a particular set of stepping conditions. For example, InstructionStepperSS is used for “Step Over” of the Other class and most instruction-level “Step In”.

|  |  |  |  |
| --- | --- | --- | --- |
| Stepper | Unit | Direction | Description |
| InstructionStepperBP | Inst. | Over | Runs to a breakpoint |
| InstructionStepperBPCall | Inst. | Over | Runs to a breakpoint after a call |
| InstructionStepperProbeCall | Inst. | In | Combined BP and SS for source code calls |
| InstructionStepperSS | Inst. | In, Over | Single steps |
| RangeStepper | Line | In, Over | Uses instruction steppers to step each instruction in a range |
| RunToStepper |  | Out | Runs to BP at a target address |

One stepper to note is InstructionStepperProbeCall. This is used for the case of stepping into a function at source level. This stepper calls back to Exec’s client to determine if stepping into a function is allowed. Typically this will involve the client checking the function for source code. If the step in is allowed, then the stepper is finished. If the step is not allowed, then we continue out of the function. This can be done in other ways, but this was the one chosen.

Steppers behave as state machines that control how a step is done. The events that cause state transitions are SS and BP exceptions that are raised in the debuggee. Each state determines an action such as: enable SS, set a BP, and make a new instruction stepper. When a stepper reaches the end state, it calls back to Exec’s client to notify of Step Complete.

Each thread in a debuggee can have a stepper. They don’t interfere with one another, because the basic operations are isolated for each thread. As explained earlier, single steps are enabled and handled for each thread. Also, using the multithreaded way of resuming from breakpoints, we effectively treat breakpoints as isolated events in a single thread. Otherwise, a BP could be hit from an unintended thread.

Steppers can be canceled. A call to Cancel makes them disable the machinery they put in place to do the step. For example, InstructionStepperBP will remove its breakpoint.

#### ResumeStepper

A Resume Stepper is an instruction level stepper used to solve the complications in stepping certain classes of instructions in order to resume from a BP. The appropriate stepper is made, stored as the Resume Stepper for the thread, and run.

Resume Steppers might duplicate or override the effort of a regular stepper, so coordination is built between the two steppers. If a thread has no regular stepper set that supports safe SS, then a Resume Stepper is used.

#### Summary

* Each thread has a stepper.
* One thread’s stepper shouldn’t affect other steppers.
* The low-level mechanisms for stepping are isolated by thread.
* Windows automatically isolates the single step mechanism by thread.
* When using temporary breakpoints to implement stepping, the breakpoints only affect one thread because all other threads are suspended.

# Debugging in Visual Studio

Visual Studio (VS) includes a plug-in called the Visual Studio Debug Package which implements the common debugger for all language environments that run in the IDE. The Debug Package implements a debugging system called AD7 (Advanced Debugging) that includes:

Components that control debug sessions

Interfaces that are implemented by Debug Engines

User interface components

When a debug session is opened, the Debug Package makes a Session Debug Manager (SDM). This object then calls into the Debug Engines to control debuggees and inspect their state. The results are presented in the common UI elements such as the call stack, memory, and disassembly panes and the Quick Watch dialog.

Debug Engines are standalone modules that are responsible for the details of certain kinds of debugging; for example, the native C++, managed code. They are implemented by third parties and have dependencies only on the VS Debug Package.

AD7 Block Structure.emf

Figure The relationship between Visual Studio, Debug Engine, and programs.

Debug Engines make up an object model of a running debuggee. They present objects such as programs, threads, breakpoints, and expressions. The VS Debug Package gets references to the objects from Debug Engines. It then calls the objects to change the state of a debuggee, control the execution, or inspect properties and data. Events raised in a debuggee are detected by the Debug Engine and communicated to the Debug Package by way of messages implemented as objects derived from a debug event interface and sent to a callback interface the Debug Package provides.

## Threading in Visual Studio

The Visual Studio Debugging threading model is laid out as follows:

* The SDM runs as a single threaded[[1]](#footnote-1) object on the main (UI) thread. From here it calls into the Debug Engine object model.
* The SDM runs the event callback on a separate thread.
* The event callback queues events it receives and dispatches them one-by-one to the SDM.
* Because event dispatch from the event callback is implemented using COM cross thread calls, the callback can still receive events while waiting for a dispatch to return.
* Debug Engines are expected to keep their own dedicated event monitoring thread. Debuggee events are sent from here to the AD7 event callback.

AD7 Threading.emf

Figure Queuing and dispatching responsibilities of SDM and Debug Engine threads.

Given the way the SDM is designed, calls from the SDM to the Debug Engine can be considered single threaded. Care then has to be taken in the interaction between the parts of the engine called by the SDM and those that run asynchronously on the event monitor thread. For the most part, this involves making cross thread calls to the event monitor thread and protecting access to certain data structures.

## LaunchSuspended, ResumeProcess, Attach

## Binding Breakpoints

Visual Studio separates the act of setting a breakpoint in a source code file from it being bound and patched at a given address of code in memory. This allows for stating the intent to break at a line in source code, even though the corresponding binary code is not loaded. Also, it allows the source code breakpoint to break at more than one address. This is the case if a library is included in more than one DLL.

As a result, the implementation of a breakpoint is split into four areas that work at different levels of abstraction.

|  |  |  |
| --- | --- | --- |
| Implementation | Area | Description |
| Pending Breakpoint | AD7 | Highest level of abstraction. Typically represents a breakpoint in source code. Contains Bound Breakpoints. Implements IDebugPendingBreakpoint2. |
| Bound Breakpoint | AD7 | Represents a Pending Breakpoint bound to a single code address. Uses one Set Breakpoint. Implements IDebugBoundBreakpoint2. |
| Set Breakpoint | Exec | Represents a handle to a Patched Breakpoint. A unique cookie is used in calls to Exec::SetBreakpoint and Exec::RemoveBreakpoint. |
| Patched Breakpoint | OS/HW | Lowest level of abstraction. Represents a physical breakpoint instruction in memory. |

As DLLs are loaded and unloaded, Bound Breakpoints are made or deleted for each code location that a Pending Breakpoint can bind to. Bound Breakpoints can be enabled or disabled individually. Enabling or disabling a Pending Breakpoint keeps the Bound Breakpoints inside it, but forces them all to become enabled or disabled.

BP Hierarchy.emf

Figure An example of dependencies in the breakpoint hierarchy.

## Disassembly

One service provided by the debug engine is a disassembly stream used by the debug package in order to fill in the Disassembly View. When scrolling thru the view, the debug package reads batches of instructions at a time from the stream. The unit for seeking and reading is instructions. Seeking the stream must be allowed forwards and backwards.

To fully enable the Disassembly View, enable the Disassembly metric for the debug engine and implement IDebugProgram2::EnumCodeContexts and IDebugProgram2::GetDisassemblyStream.

Algorithm for reading the disassembly stream when first filling the Disassembly View

Disasm Stream Seeking.emfFirst set of disassembled instructions:

1. Start at the address given to GetDisassemblyStream.
2. Seek n1 instructions backward from start address.
3. Read instructions until starting address is crossed and view is full. I’ve seen this done in batches of 30.

Scrolling backwards:

1. Seek n2 instructions back from the top of the last set of instructions read. In the case of the first backwards read, go back from the start address.
2. The same instructions we sought back we read.
3. Seek one instruction after the last one read[[2]](#footnote-2).

Going thru the disassembly stream is affected by the kind of Instruction Set Architecture. Fixed length instructions lend themselves to easy enumeration in any direction. On the other hand, architectures that have variable length instructions, like x86, complicate the process.

### x86

Because x86 uses variable length instructions, we can’t provide 100% accurate disassembly. In order to achieve decently high level of accuracy, some goals and rules are needed.

When the user asks for a disassembly, it’s with respect to a specific place in code. We declare that there is an instruction starting at that location and we want to show the user exactly what instruction it is. We call this first instruction at the starting address the anchor. Because the user assumes there is an instruction at the anchor, we consider it fully accurate. We want instructions after the anchor to be almost completely accurate, and we tolerate those before being very accurate.

In order to provide a disassembly backward from the anchor, we need a map of instruction locations and lengths that we can consult during a seek operation. We build this map one memory page at a time by reading the page, decoding each instruction, and storing their lengths at their starting addresses. All other bytes in the map are zero.

To find instructions backwards, we start at the anchor and examine each byte in the map backwards one by one. We skip bytes with value zero. And each time we read a non-zero byte, we’ve reached a new instruction.

Disasm Bufs.emf

Figure The size and location of each instruction is determined and stored.

This process provides a clear and consistent way to detect variable size instructions in memory. Even so, there are numerous details left to handle. What if the anchor is in the middle of an instruction in the map? How are instructions that cross pages handled? What happens when you add a new page to the map?

# Evaluating Expressions

A debugger would be severely hobbled without the ability to evaluate user provided expressions expressed in the very language being debugged in order to determine their run time value.

The Mago Expression Evaluator computes the value of an expression in the D programming language. Expressions are provided as text strings; and the results are provided numeric values or addresses. Identifiers, types, and references values are resolved thru an external interface. In this way, the evaluator does not depend on any debug format or runtime environment. Indeed, in the unit tests declarations and input values are written in XML.

The Expression Evaluator is also responsible for formatting values, and enumerating children of values.

## Scanner and Parser

As in a compiler, the scanner and parser transform text into a structured set of objects representing the meaning of the original text. The scanner recognizes groups of characters and turns the text into a stream of tokens. The parser recognizes structured groups of tokens and produces the node tree.

EE Flow.emf

Figure The flow of data in the Expression Evaluator from text characters to the result value.

In order to focus on the task of the expression evaluator, the parts of the language spec dealing with the syntax of declarations and statements were left out. Only expression parsing is implemented.

## Node Tree

Nodes are objects representing the individual elements and structures of the language:

* Multiplying two expressions
* Indexing an array
* Getting the address of a variable
* A variable

Each node represents an individual object with a value or an operation. They are arranged to refer to each other according to the structure of the expression in the original input text string. Nodes are given two responsibilities:

1. Bind to declarations, types, and values. The node takes on the properties of the system the evaluator is mimicking.
2. Evaluate. Compute the value of the node by first evaluating child nodes, and then combining the results according to the meaning of the node.

Some evaluations are simple. An Integer node corresponding to the expression “3” will evaluate to the integer 3. Other evaluations are much more complex. If function evaluation were allowed, then it would involve manipulating the debuggee to call a single function.

## Declarations, Types, and the Binder

The evaluator declares several interfaces, some of which are meant to be used by the client of the evaluator. These interfaces are used to customize the parts of the expression.

Named elements like variables and certain types are described by declarations. IDeclaration gives clients a way to specify attributes like the address of a variable, the members of a struct, or the name of a typedef.

Types are described by classes in the hierarchy rooted in the class Type. Since there are few ways to customize a type, the whole Type tree is implemented in the evaluator. Certain kinds of types are customized thru the use of IDeclaration. This allows specifying the members of a struct or class.

The TypeEnv (Type Environment) is responsible for making new instances of type objects. By providing this class, we can keep the Type tree implementation internal to the evaluator. TypeEnv offers methods for instantiating correctly initialized type objects, for example NewSArray( Type\* elem, uint32\_t length, Type\*& type ) for static arrays.

Finally, IValueBinder is implemented by clients and used by the evaluator to look up declarations by name; get and set values for an address or declaration; and reading memory.

## Other Features

In order to keep all functionality related to the D programming language in one place, the Mago Expression Evaluator (EE) takes on some more features.

The EE knows how to read text in D, so it’s fitting that it knows how to write value in D. The EE can format numbers, strings, arrays, and other values how they are represented in D or in a way that’s consistent with the language.

Another feature related to the language is enumerating the children of values. Static arrays have a fixed number of elements declared in the type itself. Dynamic arrays have the number of elements in values of the type. The EE can take values of types such as these and others and list related values.

|  |  |
| --- | --- |
| Type | Children |
| Pointer | Pointed value |
| Static and Dynamic Arrays | Elements |
| Struct and Class | Fields |

# Debug Information

What makes stepping, breakpoints, and expression evaluation work for a particular program are the details about the program provided by debug information (Debug Info). Without debug information, the debugging experience would be limited to assembly level stepping and evaluating the value of hardware registers and memory.

Debug Info can be embedded with the program or stand on its own in a separate file. Depending on the system, it comes in different formats: PDB, stabs, DWARF, CodeView. Usually, Debug Info is language agnostic; though some formats are more suitable to one kind of programming language than others.

Details that are typically found in debug info are:

* Maps of source files to lines
* Maps of source code lines to addresses and back
* Functions – their addresses and scopes
* Symbols, including their name, type, value, and storage details
* Types, including their relationship to other types

## CodeView

CodeView is the debug info format made for the Microsoft CodeView debugger from the 80’s and 90’s. It comes in different versions and two flavors: 16 and 32-bit. The latest version for which a specification was publicly released is version 4.10. The DMD compiler on Windows produces debug info in the 32-bit flavor of this format.

Notable features of CodeView 4.10 include:

* **Fixed record fields.** For example, a register symbol record will always have a type field at byte 4 in the record.
* **Numeric constant compression.** Values less than 32768 can be stored in two bytes without identification. Otherwise, a tag is used before the value.
* **Common type encoding.** Types are assigned 16-bit indexes. An index maps from a type index number to a type record. Common types like integers and characters have no type record, but are identified directly by their index number.
* **Sorted symbols.** Special sections of debug info are reserved for tables of sorted symbols by name and address.
* **Lexical blocks** can nest inside functions.

# Appendix

## Stepping Scenarios by Instruction Class

### Simple Instructions

Stepping Detail - Simple.emf

### Call Instructions

Stepping Detail - Call.emf

### Rep Instructions

Stepping Detail - Rep.emf

# References and Links

Mago Debugger

<http://dsource.org/projects/mago_debugger>

D Programming Language

<http://digitalmars.com/d/>

Visual Studio Debugger Extensibility

<http://msdn.microsoft.com/en-us/library/bb161718.aspx>

Intel® 64 and IA-32 Architectures Software Developer's Manuals

<http://www.intel.com/products/processor/manuals/>

CodeView 4.10 Spec

<http://dsource.org/projects/mago_debugger/browser/docs/Reference/MS_Symbol_Type_v1.0.pdf>

<http://pierrelib.pagesperso-orange.fr/exec_formats/index.html>

Udis86 Disassembler for x86 and x64

<http://udis86.sourceforge.net/>

1. SDM is a Single Threaded Apartment (STA) COM object living in the main (UI) thread. If the DE is also STA, then it will live in the main thread. Otherwise, it will live in an RPC worker thread. [↑](#footnote-ref-1)
2. I believe this is for verifying that the ranges are continuous and don’t have gaps. [↑](#footnote-ref-2)