Develop A Golang Debugger

Author: ZhangJie

Changelog

|  |  |  |
| --- | --- | --- |
| **提交时间** | **修改内容** | **备注** |
| 2018-06-18 | 内容大纲 |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Debugger Basic

## Purpose

Though programmers pay lots of attention to code bug free programs, making bugs cannot be avoided. Print statements are often used to locate the bugs, but in some complex occasions, we need to control the execution of tracee (debugged process) and inspect its runtime state including memory, registers.

This book aims to guide us to develop a golang debugger, so how to use a debugger is put on the second burner. But if you have any experience in debugging using symbol debuggers, you can understand the implementation details much more easily.

Common debugging operations are as following:

1. Set breakpoint on memory address, function, statement, file line number
2. Single step instruction, single step statement, or continue to breakpoint
3. Get/Set registers info
4. Get/Set memory info
5. Evaluate expressions
6. Call statement
7. Others

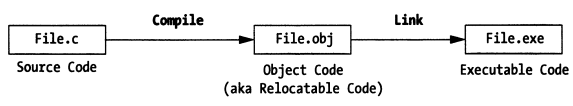
This book will describe how to implement the relevant debugging operations. If you are interested in this boring debugging internals, then keep going on.

## Dependencies

### Debug Symbol Info

The compiler and linker build the executable file based on source code. The data of executable file is generated for machine, rather than for human. How does a source level debugger understand that data and remap it to source information or vice versa?

This depends on the **debug symbol table**. When the compiler converts source code into object file, it will also generate the self-contained debug symbols. When the linker links all object files into the executable file, it will merge the debug symbols stored in these object files into the debug symbol table.



There’s some standards to guide the compiler to generate debugging symbols to coordinate the compiler, linker and debugger, such as Dwarf. Compiler and linker generate these debugging sections and store them into the executable file, debugger can extract these sections to build a source level view, then debugger can do some remapping task between memory address, instruction address and source code.

>>>Remark:

In practice, depending on the format of the object file, debug symbol table records are typically placed in one of two locations:

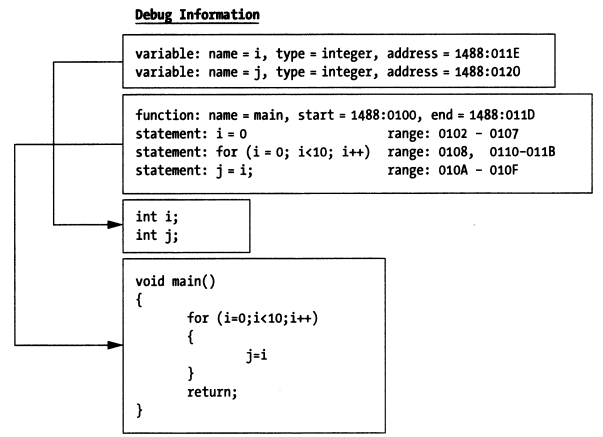
* In the body of the object code itself

For example, ELF object format contains Dwarf debug symbol table.

* In a separate file

For example, Microsoft’s Visual C++ 2.0 debug information is stored in a separate \*.PDB (Program Database) file.

Debug symbol information maps functions and variables to locations in memory, this is what gives a symbolic debugger the fundamental advantage over a machine debugger. For instance, the source code to memory mapping allows a symbolic debugger to display the value of a variable, because the variable’s identifier is matched to a specific location in the program’s data segment (stack or heap). Not only that, but there will also be data type information in the symbol table that will tell the debugger what type of data is manipulated so that its value can be properly displayed.



This mapping also matches source code statements to ranges of bytes in memory. When you step into a source code statement, the symbolic debugger will look up the address range of the given statement in the program’s debug records. Then it will simply execute the machine instructions in that range.

### Debug Infrastructure

Besides the debug symbol info, some other dependencies are still needed, i.e., the debug infrastructure, including debugging interrupts, system calls, interpreters, debug interface (GUI or command-line).

#### Debugging Interrupts

All of the commercial operating systems provide hooks for debugging. These hooks are usually implemented as system calls inside of the kernel. This is as necessary because debugging an application requires access to system data structures that exist in a protected region of memory, i.e., the kernel. The only way to manipulate these special data structures is to politely ask the operating system to do so on your behalf.

One exception to this rule occurs in the case of DOS. With DOS, a real mode operating system, you can do damn nearly everything by yourself because memory protection does not exist.

#### System Calls

Nowadays, most operating systems implement memory’s protective mode, it is the base stone of multi-user, multi-task operating system. If there’s no protective mode, there’s no so-called security at all.

Opposite to DOS, Windows, Linux, BSD have fairly sophisticated memory protection scheme. This means that if you want to write a debugger, you’ll need to rely on the system calls.

Take Linux system calls as an example, the tracee process can be attached via **ptrace(PTRACE\_ATTACH…)**, then the tracee will be notified by **SIGSTOP** sent by kernel, then tracee will stop, tracer process can call **waitpid(pid)** to wait this happens. After that, tracer process can call ptrace with other request param (PTRACE\_GETREGS, PTRACE\_SETREGS, PTRACE\_PEEKDATA, PTRACE\_POKEDATA…) to further inspect the tracee runtime state and control its code execution path.

#### Interpreters

As regards with debugging an interpreted language, it is much more direct than the system call approach because all of the debugging facilities can be built directly into the interpreter. Within an interpreter, you have unrestricted access to the execution engine. The entire thing can run in user space instead of kernel space (syscall). Nothing is hidden. All you need to do is add extensions to process breakpoint instructions and support single stepping.

#### Kernel Debuggers

When an operating system institutes strict memory protection, a special type of debugger is needed to debug the kernel. You cannot use a conventional user-mode debugger because memory protection facilities (like segmentation and paging) prevent it from manipulating the kernel’s image.

Instead, what you need is a kernel debugger.

A kernel debugger is an odd creature that commandeers control the processor so that the kernel can be examined via single stepping and breakpoints. This means that the kernel debugger must somehow sidestep the native memory protection scheme by merging itself into the operating system’s memory image. Some vendors perform this feat by designing their debuggers as device drivers, or loadable kernel modules.

#### Debug Interface

In case you haven’t noticed, it’s all about program state. Different debuggers offer different ways for a user to view the state of a running program. Some debuggers, like gdb, provide only a simple, but consistent, command-line interface. Other debuggers are integrated into slick GUI environments. To be honest, I lean towards the GUI debuggers because they are capable of presenting and accessing more machine state information at any given point in time. With a GUI debugger, you can easily monitor dozens of program elements simultaneously.

On the other hand, if you are developing an application that will be deployed on multiple platforms, it may be difficult to find a GUI IDE that runs on all of them. This is the great equalizer for command-line debuggers. The GNU debugger may not have a fancy interface, but it looks (and behaves) the same everywhere. Once you jump the initial learning curve, you can debug executables on any platform that gdb has been ported to.

### Symbol Debugger Extensions

#### Dynamic Breakpoints

If there’s a term called dynamic breakpoints, there may be a term called static breakpoints. Yes, both of them exist.

Breakpoints are created by generating 0xCC one-byte machine instruction, 0xCC causes processor to pause the running process. If you write assembly, int 3 can be used to generate this instruction 0xCC。After understanding purpose of 0xCC, we can continue discussing the breakpoints’ types, the static breakpoints and the dynamic breakpoints.

1. **Static breakpoints**

Static breakpoints refers to the breakpoints generated by “int 3” assembly which are programmatically inserted into the program source code. These breakpoints’ lifetime is as the same of this process. We can insert branch control logic, which can be enabled or disabled by arguments, to determine whether specific breakpoints are enabled or not.

Some assembly instruction for getting/setting memory/registers can also be inserted.

Better solution is to encapsulate the relevant assembly operation into a library, which can be linked and used conveniently for any other programs.

1. **Dynamic breakpoints**

In the previous part, I used static breakpoint instructions that were manually inserted at compile time. An alternative to this approach is to dynamically insert breakpoints into a program’s memory image at runtime. As you will see later on, this allows symbolic debuggers to single-step through a program at the source code level.

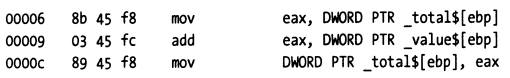
Unlike static breakpoints, which exist for the duration of a program’s lifecycle, symbolic debuggers usually work with dynamic breakpoints. The insertion, and removal of dynamic breakpoints obyes the following scheme:

* The debugger identifies the first opcode of a statement
* The debugger saves the opcode and replaces it with a breakpoint (0xCC)
* The debugger digests the breakpoint and halts execution
* The debugger restores the original opcode
* The debugger leaves the opcode or swaps in another breakpoint

Let’s take the following statement in C as an example:

Total = total +value;

Providing the associated assembly is as following:



To place a dynamic breakpoint on a statement, the debugger would take the first opcode 0x8B and replace it with a breakpoint instruction 0xCC. When the debugger encounters this breakpoint, it will replace the breakpoint with the opcode and then execute the entire statement.

Once the statement has been executed, the debugger then has the option to swap back in the breakpoint or to leave the instruction alone. If the breakpoint was originally inserted via an explicit request by the user (i.e., break source.c:17), it will be reinserted again. However, if the breakpoint was initially inserted to support single stepping, the breakpoint will not be reinserted.

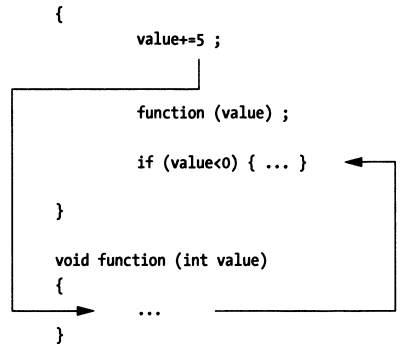
#### Single Stepping

Single stepping in a machine-level debugger is simple: the processor simply executes the next machine instruction and returns program control to the debugger. For a symbolic debugger, this process is not as simple because a single statement in a high-level programming language typically translates into several machine-level instructions. You can’t simply have the debugger execute a fixed number of machine instructions because high-level source code statements vary in terms of how many machine-level instructions they resolve to.

To single-step, a symbolic debugger has to use dynamic breakpoints. The nature of how dynamic breakpoints are inserted will depend upon the type of single stepping being performed. There are three different types of single stepping:

1. **Single stepping into (the next statement)**

When a symbolic debugger steps into a source code statement, it scans the first few machine instructions to see if the statement is a functions invocation. If the first opcode of the next instruction is not part of a function invocation, the debugger will simply save the opcode and replace it with a breakpoint. Otherwise, the debugger will determine where the function invocation jumps to, in memory, and replace the first opcode of the function’s body with a breakpoint such that execution pauses after the function has been invoked.

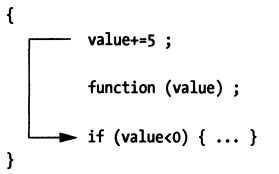


1. **Single stepping out of (a routine)**

When a source-level debugger steps out of a routine, it looks through the routine’s activation record for a return address. It then saves the opcode of the machine instruction at this return address and replaces it with a breakpoint. When program execution resumes, the routine will complete the rest of its statements and jump to its return address. The execution path will then hit the breakpoint, and program control will be given back to the debugger. The effect is that you are able to force the debugger’s attention out of a function and back to the code that invoked it.

1. **Single stepping over (the next statement)**

When a source-level debugger steps over a statement, it queries the program’s symbol table to determine the address range of the statement in memory (this is one scenario in which the symbol table really comes in handy). Once the debugger has determined where the statement ends, it saves the opcode of the first machine instruction following the statement and replaces it with a breakpoint. When execution resumes, the debugger will regain program control only after the path of execution has traversed the statement.

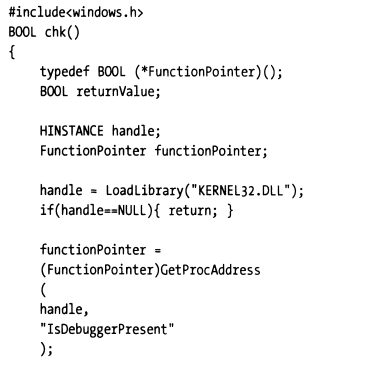


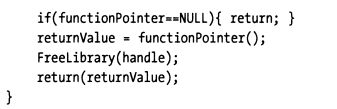
## Countertactics

Give enough time and effort, any program can be reverse engineered. The goal, then, is to make it as painful as possible for a malicious engineer to figour out how things work. In light of this, there’re steps that you can take that will make it difficult for someone to peek at your program with a debugger. In this section, I will examine a few of these steps.

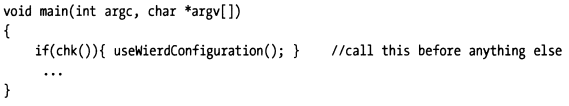
### System Calls

Some operating systems provides a special call that will indicate if the current process is being executed under the auspices of a debugger. For example, Windows KERNEL32.DLL exports a function named IsDebuggerPresent(). You can wrap this call in an innocuous little routine like chk().





The trick to this technique is to call chk() immediately. This will increase the likelihood that the code will get a chance to execute before the debugger encounters the first breakpoint.



If a debugger is present, you can force the program to behave strangely, and send the person debugging your application on a wild-goose chase. Debuggers are unique tools because they allow the user to observe a program from a neutral frame of reference. By inserted code like chk(), you are forcing the user into a warped quantum universe where the very act of observing influences the output of the program.

>>>Remark

In Linux, we may use other similar tricks to determine whether current running process is debugged or not.

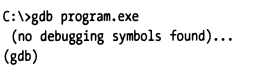
### Remove Debugging Information

One simple way to make debugging more expensive is to remove debugging information from your deliverable. This can be done by stripping debug information (with a tool like GNU’s strip utility) or by setting your development tools to generate a release build.

Some business software companies prefer to strip debug information and accept the associated performance hit, because it allows sales engineers to perform an on-site diagnosis. When sales engineers make a house call, all that they need to do in order to take a look under the hood is insert the debug information and crank up a debugger.

The gcc compiler uses the **option -g** to insert debug information in the object code that it generates. If the option is not specified, then no symbol information will be included for debugging purposes.

If you try and debug this with gdb, it will complain that it cannot find any debugging symbols. The absence of debugging symbols will make it very difficult to see what’s going on in terms of anything but raw machine state.



The absence of debugging symbols will not stop everyone, some decompilers out there can take machine code and recast it as high-level source code. The good news is that these tools tend to generate code that is difficult to read and use arbitrary naming conventions. In other words, the cure is almost as bad as the illness.

### Coding Salt

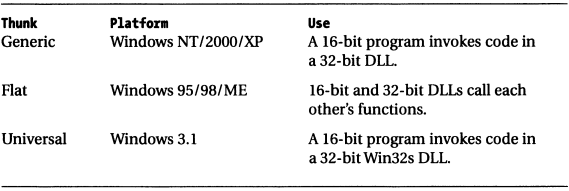
If memory footprint is not a big issue, and you don’t mind a slight performance hit, one way to foil a debugger is to periodically salt your code with unnecessary statements. This will make it easy for someone trying to reverse engineer your code to become lost among the trees and lose sight of the forest, so to speak.

In this manner, even if you shipped this program with debug symbols, it would be difficult to figure out what was happening (particularly if you believed that each statement had a legitimate purpose).

### Mixed Memory Models

There’re robust debuggers, like SoftICE, that can gracefully make the jump between user mode and kernel mode. However, not many debuggers can make the jump between two different memory models. Windows in particular is guilty of allowing this kind of abomination to occur. On Windows, this phenomenon is generally known as thunking, and it allows 16-bit code and 32-bit code to fraternize.

Folliwing depicts the thunking techniques used in Windows:



# Dwarf

DWARF is a widely used, standardized debugging data format. DWARF was originally designed along with Executable and Linkable Format (ELF), although it is independent of object file formats. The name is a medieval fantasy complement to "ELF" that has no official meaning, although the backronym '**Debugging With Attributed Record Formats**' was later proposed.

## History

The first version of DWARF proved to use excessive amounts of storage, and an incompatible successor, DWARF-2, superseded it and added various encoding schemes to reduce data size. DWARF did not immediately gain universal acceptance; for instance, when Sun Microsystems adopted ELF as part of their move to Solaris, they opted to continue using **stabs, in an embedding known as "stabs-in-elf"**. Linux followed suit, and DWARF-2 did not become the default until the late 1990s.

The DWARF Workgroup of the Free Standards Group released DWARF version 3 in January 2006, adding (among other things) support for C++ namespaces, Fortran 90 allocatable data and additional compiler optimization techniques.

The DWARF committee published version 4 of DWARF, which offers "improved data compression, better description of optimized code, and support for new language features in C++", in 2010.

Version 5 of the DWARF format was published in February 2017. It "incorporates improvements in many areas: better data compression, separation of debugging data from executable files, improved description of macros and source files, faster searching for symbols, improved debugging of optimized code, as well as numerous improvements in functionality and performance."

## Dwarf 2

### structure

DWARF uses a data structure called a Debugging Information Entry (DIE) to represent each variable, type, procedure, etc. A DIE has a tag (e.g., DW\_TAG\_variable, DW\_TAG\_pointer\_type, DW\_TAG\_subprogram…) and set of attributes (key-value pairs). A DIE can have nested (child) DIEs, forming a tree structure. A DIE attribute can refer to another DIE anywhere in the tree—for instance, a DIE representing a variable would have a DW\_AT\_type entry pointing to the DIE describing the variable's type.

**To save space, two large tables needed by symbolic debuggers are represented as byte-coded instructions for simple, special-purpose finite state machines.**

1. **The Line Number Table**, which maps code locations to source code locations and vice versa, also specifies which instructions are part of function prologues and epilogues.
2. **The Call Frame Information table**, which allows debuggers to locate frames on the call stack.

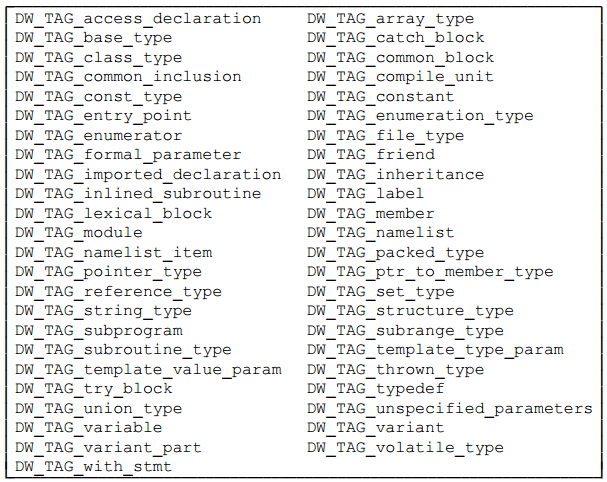
### DIE

Each debugging information entry is described by **an identifying tag** and contains **a series of attributes**. The tag specifies the class to which an entry belongs, and the attributes define the specific characteristics of the entry.

The debugging information entries in Dwarf v2 are intended to exist in the **.debug\_info** section of an object file.

#### Tag

Tag, specifies what the DIE describes, the set of required tag names is listed in following figure.



#### Attribute

Attribute, fill in details of DIE and further describes the entity.

An attribute has a variety of values: constants (such as function name), variables (such as start address for a function), or references to another DIE (such as for the type of functions’ return value).

The permissive values for an attribute belong to one or more classes of attribute value forms. Each form class may be represented in one or more ways.

For instance, some attribute values consist of a single piece of constant data. “Constant data” is the class of attribute value that those attributes may have. There’re several representations of constant data, however (one, two, four, eight bytes and variable length data). The particular representation for any given instance of an attribute is encoded along with the attribute name as part of of the information that guides the interpretation of a debugging information entry.

The set of required attribute names is listed in following figure.



*>>>Remark:*

*Attribute value forms may belong to one of the following classes:*

1. *Address, refers to some location in the address space of the described program.*
2. *Block, an arbitrary number of uninterpreted bytes of data.*
3. *Constant, one, two, four or eight bytes of uninterpreted data, or data encoded in LEB128.*
4. *Flag, a small constant that indicates the presence or absence of the an attribute.*
5. *Reference, refers to some member of the set of DIEs that describe the program.*
6. *String, a null-terminated sequence of zero or more (non-null) bytes. Strings maybe represented directly in the DIE or as an offset in a separate string table.*

#### Form

Briefly, DIE can be classified into 2 forms:

1. the one to describe the data type
2. the one to describe the function and executable code

One DIE can have parent, siblings and children DIEs, dwarf debugging info is constructed as a tree in which each node is a DIE, several DIE combined to describe a entity in programming language (such as a function).

In following sections, types of DIEs will be described before we dive into dwarf further.

### Describing Data and Type

Most of programming languages provide sophisticated description of data types, including the builtin basic data type and the method to create new data type. Dwarf aims to support all programming languages, so it abstracts out a solution to represent all languages.

Dwarf abstracts out several base types (number) based on the hardware, other types are constructed as collections or composition of base types.

#### Base Types

Dwarf v1 and other debugging formats suppose that compiler and debugger need sharing common understanding of the size of base type, such as whether an int is 8, 16 or 32 bits. Same language on different hardware platforms may have different size of the same type, an int could be 16 bit on 16 bit processor and 32 bit on 32 bit processor. But different language on the same hardware platform may have different size of the same type, such as golang int is 64 bit on 64 bit processor, while in C it is 32 bit.

So the problem is how to remapping the base type to different bit size flexibly? Dwarf v2 solves this issue, it provides the lowest level mapping between the simple data types and how they are implemented on the target machine’s hardware.

Here is 3 examples.

Figure 2a depicts that type int is 4 byte signed numer on 32-bit processor, figure 2b depicts that int is 2 byte signed number on 16-bit processor.

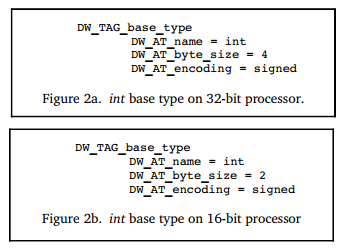
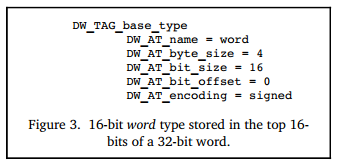


figure 3 depicts that type word is 16 bit signed number, while it takes up 4 bytes, in which only the first 2 upper bytes is used, leading the lower 2 bytes zero.

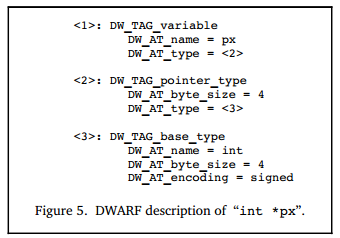


DW\_TAG\_base\_type, this tag allows a number of base types to be described, including binary integer, packed integer, address, character, fixed number and floating number. floating number’s encoding format (such as IEEE-754) is determined by the hardware.

#### Type Composition

Dwarf defines a new data type via grouping or chaining builtin base types.

Figure 5 depicts that variable px is defined, its data type DW\_AT\_type = <2> references DW\_TAG\_pointer\_type, this pointer byte size is 4 byte, this pointer type is defined via modifying base type int (DW\_AT\_type = <3>), which is defined by DW\_AT\_base\_type.



Other data types can also be defined via chaining several DW\_TAG\_...., such as reference type can be built based on DW\_TAG\_pointer\_type.

#### Array

Array is defined in a DIE which specifies whether the elements is stored in column major order (as in Fortran) or row major order (as in C or C++). The index for the array is specified by a subrange type that gives the lower and upper bounds of each dimension. This allows dwarf to describe both C style arrays, which always have zero as the lowest index, as well as arrays in Pascal or Ada, which can have any value for the low and high bounds.

#### Structures, Classes, Unions, and Interfaces

Most programming languages allow programmer to group different data types into a new compositive type, such as struct.

Struct allows grouping several members of different types. C union allows this too, but every member shares the same space. C++ struct add some features beyond of C. C++ class and Java interfaces are similar entities to some extent.

Different languages may have nearly the same compositive data type, but data type name differs, such as C++ class and class members are called Record and Fields in Pascal, Dwarf uses the terminology in C\C++\Java.

DIE for a class is a parent for DIEs of class members, each class has a name and other possible attributes. If the size of class instance is known at compile time, the DIE will have a DW\_AT\_byte\_size attribute. Each of these descriptions are very similar to a base type, which may be added some more details, such as class members accessibility modifier.

C\C++ also allow bit field in struct, this can be described in attribute DW\_AT\_bit\_offset

and DW\_AT\_bit\_size combined with DW\_AT\_byte\_size. Bit offset and bit size specify

how many bits is taken up and where it is stored, byte size specifies total space taken

up by the struct.

#### Variables

Variables are generally pretty simple, they have a name which presents a chunk of memory (or register) that store the value. Variable’s type describes what value is contained and whether it can be modified (such as const) or not.

What distinguishes a variable is where its value is stored and its scope.

1. A variable can be stored at global data section, stack or register.
2. Variable’s scope describes where it can be known in the program, to some extent, its scope is determined by declaration. Dwarf documents where the variable is defined in triplet (file, line, column).

#### Location Expressions

##### Introduction

Dwarf provides a very general schema to describe how to locate the data represented by a variable. That is Dwarf location attribute DW\_AT\_location, which specifies a sequence of operations to tell debugger how to locate the data.

Following is an example to show how DW\_AT\_location attribute helps to locate the variable address.

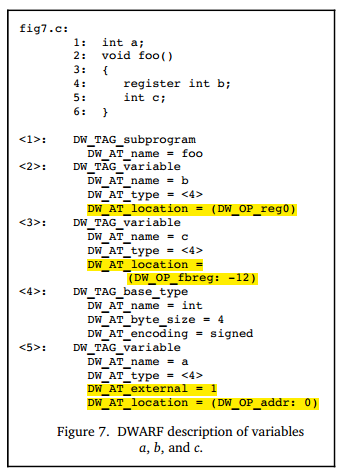


Figure 7 depicts that variable b is stored in a register, variable c is stored at stack, variable a is stored at fixed address (data section).

##### Further Reading

*The debugging information must provides consumers a way to find the location of program variables, determine the bounds of of dynamic arrays and strings and possibly to find the base address of a subroutine’s stack frame or the return address of a subroutine. Furthermore, to meet the needs of recent computer architectures and optimization techniques, the debugging information must be able to describe the location of an object whose location changes over the objects’ lifetime.*

*Information about the location of program object is provided by location descriptions. Location descriptions can be classified into two forms:*

1. *Location expressions, which are a language independent representation of addressing rules of arbitrary complexity built from a few basic building blocks, operations. They are sufficient for describing the location of any object as long as its lifetime is either static or the same as the lexical block that owns it, and it does not move throughout its lifetime.*
2. *Location lists, which are used to describe objects that have a limited lifetime or change their location throughout their lifetime.*

##### Location Expression

*A location expression consists of zero or more location operations. An expression with zero operations is used to denote an object that is present in the source code but not present in the object code (perhaps because of optimization).* ***The location operations fall into two categories, register names and addressing operations.***

1. ***Register names*** *always appear alone and indicate that the referred object is contained inside a particular register.*

*Note that the register number represents a Dwarf specific mapping of numbers onto the actual registers of a given architecture. DW\_OP\_reg${n} (0<=n<=31) operations encode the names of up to 32 register, the object is addressed in register n. DW\_OP\_regx operation has a single unsigned LEB128 literal operand that encodes the name of a register.*

1. ***Address operations*** *are memory address computation rules. All location operations are encoded as a stream of opcodes that are each followed by zero or more literal operands. The number of operands is determined by the opcode.*

*Each addressing operation represents a postfix operation on a simple stack machine. Each element of the stack is the size of an address on the target machine. The value on the top of the stack after executing the location expression is taken to be the result (the address of the object, or the value of the array bound, or the length of a dynamic string). In the case of locations used for structure members, the computation assumes that the base address of the containing structure has been pushed on the stack before evaluation of the address operation.*

*There’re several address operation manners, including:*

1. ***Register Based Addressing***

*Register based addressing, push a value onto the stack that is the result of adding the contents of a register with a given signed offset.*

*DW\_OP\_fbreg $offset, adding contents in frame base register (rbp) with $offset.*

*DW\_OP\_breg${n}, adding contents in register ${n} with LEB128 encoded offset.*

*DW\_OP\_bregx, adding contents in register whose number is LEB128 encoded with a LEB128 encoded offset .*

1. ***Stack Operations***
2. ***The following operations all push a value onto the addressing stack:***

*DW\_OP\_lit${n} (0<=n<=31), encode the unsigned literal values ${n}.*

*DW\_OP\_addr, encode the machine address that matches the target machine.*

*DW\_OP\_const1u/1s/2u/2s/4u/4s/8u/8s, encode 1/2/4/8 bytes unsigned or signed integer.*

*DW\_OP\_constu/s, encode LEB128 unsigned or signed integer.*

1. ***Following operations manipulate the location stack, location operations that index the location stack assumes that the top of the stack has index 0.***

*DW\_OP\_dup, duplicates the top stack entry and pushes.*

*DW\_OP\_drop, pops the value at the top of stack.*

*DW\_OP\_pick, picks the stack entry specified by 1-byte ${index} and pushes.*

*DW\_OP\_over, duplicate the stack entry with index 2 and pushes.*

*DW\_OP\_swap, swap two stack entries, which are specified by two operands.*

*DW\_OP\_rot, rotate the top 3 stack entries.*

*DW\_OP\_deref, pops the value at the top of stack as address and retrieves data from that address, then pushes the data whose size is the size of address on target machine.*

*DW\_OP\_deref\_size, similar to DW\_OP\_deref, plus when retrieveing data from address, bytes that’ll be read is specified by 1-byte operand, the read data will be zero-extended to match the size of address on target machine.*

*DW\_OP\_xderef & DW\_OP\_xderef\_size, similar to DW\_OP\_deref, plus extended dereference mechanism. When dereferencing, the top stack entry is popped as address, the second top stack entry is popped as an address space identifier. Do some calculation to get the address and retrieve data from it, then push the data to the stack.*

1. ***Arithmetic and Logical Operations***

*DW\_OP\_abs, DW\_OP\_and, DW\_OP\_div, DW\_OP\_minus, DW\_OP\_mod, DW\_OP\_mul, DW\_OP\_neg, DW\_OP\_not, DW\_OP\_or, DW\_OP\_plus, DW\_OP\_plus\_uconst, DW\_OP\_shl, DW\_OP\_shr, DW\_OP\_shra, DW\_OP\_xor, all these operations works similarly, pop the operands from the stack and calculate, then push value to the stack.*

1. ***Control Flow Operations***

*The following operations provide simple control of flow of a location expression.*

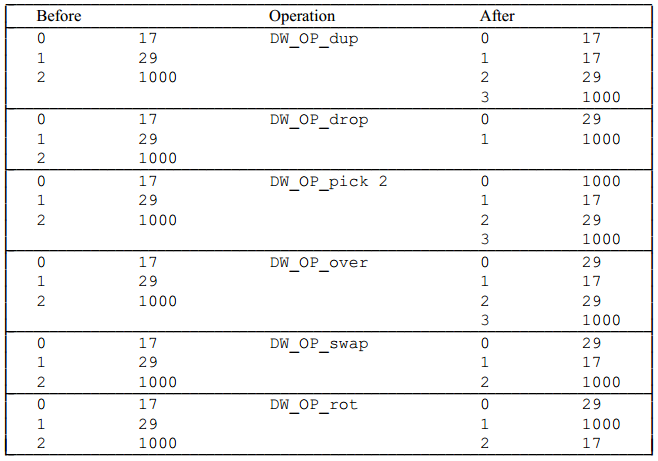
1. *Relational operators, the six operators each pops the top two stack entries and compares the top first one with the second one, and pushes value 1 if the result is true or pushes value 0 if the result is false.*
2. *DW\_OP\_skip, unconditional branch, its operand is a 2-byte constant representing the number of bytes of the location expression to skip from current location expression, beginning after the 2-byte constant.*
3. *DW\_OP\_bra, conditional branch, this operation pops the stack, if the popped value is not zero, then skip some bytes to jump to the location expression. The number of bytes to skip is specified by its operand, which is a 2-byte constant representing the number of bytes of the location expression to skip from current locating expression, beginning after the 2-byte constant.*
4. ***Special Operations***

*There’re two special operations currently defined in Dwarf 2:*

1. *DW\_OP\_piece, many compilers store a single variable in a set of registers, or store partially in register and partially in memory. DW\_OP\_piece provides a way of describing how large a part of a variable a particular address location refers to.*
2. *DW\_OP\_nop, it’s a placeholder, it has no effect on the location stack or any of its values.*

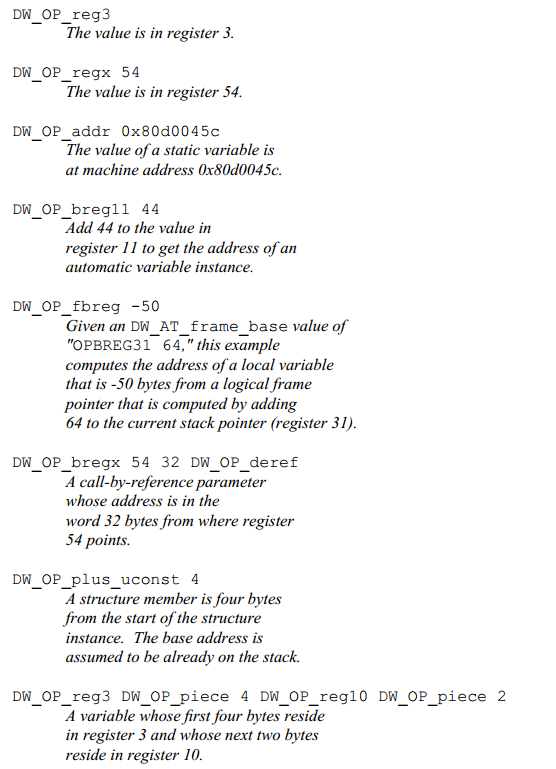
The location operations mentioned above are described conventionally, following are some examples.

1. Stack Operation Sample



1. Location Expression Sample

Here are some examples of how location operations are used to form location expressions.



##### Location Lists

Location lists are used in place of location expressions whenever the object whose location can be changed during its lifetime. Location lists are contained in a separate object file section .debug\_loc.

A location list is indicated by a constant offset from the beginning of the .debug\_loc to the first byte of this list for the object in question.

Each entry in location list consists of:

1. A beginning address, it is relative to the base address of the compilation unit referring to this location list, it marks the beginning of the address range over which the location is valid.
2. An ending address, it is again relative to the base address of the compilation unit referring to this location list, it marks the end of the address range over which the location is valid.
3. A location expression, it describes the location of the object over range specified by the beginning and end address.

The end of any location list is marked by a 0 for the beginning address and a 0 for the end address, no location description is provided.

*Remark: I’m a little confused with this, and Dwarf 5 will replace .debug\_loc and .debug\_ranges with .debug\_loclists and .debug\_rnglists allowing more compact representation and eliminating relocations.*

#### Further Reading

1. Types of Declarations, please refer to 3.2.2.1 and 3.2.2.2.
2. Accessibility of Declarations, some languages provides support for accessibility of an object or some other program entity, this can be specififed by attribute DW\_AT\_accessibility, whose value is a constant drawn from the set of codes listed here: DW\_ACCESS\_public, DW\_ACCESS\_private, DW\_ACCESS\_protected.
3. Visualbility of Declarations, it specifies which declaration are to be visible outside of the module in which they are declared, this can be specified by attribute DW\_AT\_visualbility, whose value is constant drawn from the set of codes listed here: DW\_VIS\_local, DW\_VIS\_exported, DW\_VIS\_qualified.
4. Virtuality of Declarations, C++ provides support for virtual and pure virtual structure or class member functions and for virtual base classes, this is specified by attribute DW\_AT\_virtuality, whose value is a constant drawn from codes listed here: DW\_VIRTUALITY\_none, DW\_VIRTUALITY\_virtual, DW\_VIRTUALITY\_pure\_virtual.
5. Artificial Entries, a compiler may wish to generate debugging information entries for objects and types that were not actually declared in the source of the application. An example is a formal parameter entry to represent the hidden ***this*** parameter that most C++ implementations pass as the first argument to non-static member functions.
6. Declaration coordinates, any DIE representing the declaration of an object, module, subprogram or type may have DW\_AT\_decl\_file, DW\_AT\_decl\_line and DW\_AT\_decl\_column attributes, each of whose value is a constant.

### Describing Executable Code

#### Functions and SubPrograms

1. subprogram

Functions (also called subprogram) may has return value or not, Dwarf use DIE DW\_AT\_subprogram to represent both these two cases. This DIE has a name, a source location triplet, and an attribute which indicates whether the subprogram is external, that is, visible outside the current compilation unit.

1. subprogram address range

A subprogram DIE has attributes DW\_AT\_low\_pc, DW\_AT\_high\_pc to give the low and high memory addresses that the subprogram occupies. In some cases, maybe the subprogram memory address is continuous or not, if not, there’ll be a list of memory ranges. The low pc is assumed to be the entry point of subprogram unless another one is specified explicitly.

1. subprogram return type

A subprogram’s return value’s type is described by the attribute DW\_AT\_type within DIE DW\_TAG\_subprogram. If no value returned, this attribute doesn’t exist. If return type is defined within the same scope with this subprogram, the return type DIE will also be an sibling of this DIE subprogram.

1. subprogram formal parameters

A subprogram may have zero or several formal parameters, which are described by DIE DW\_TAG\_formal\_parameter, will be listed after the DIE subprogram as the same order as declared in parameter list, though DIE of parameter type may be interspersed. Mostly, these formal parameters are stored in registers.

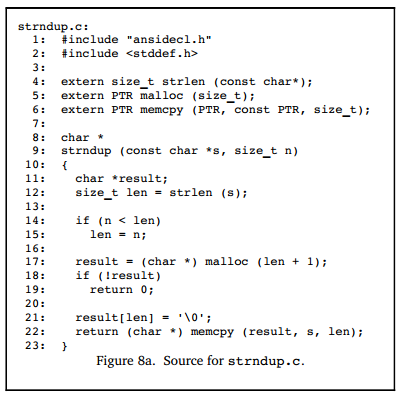
1. subprogram variables

A subprogram body may contains local variables, these variables are described by DIE DW\_TAG\_variables listing after formal parameters’ DIEs. Mostly, these local variables are allocated in stack.

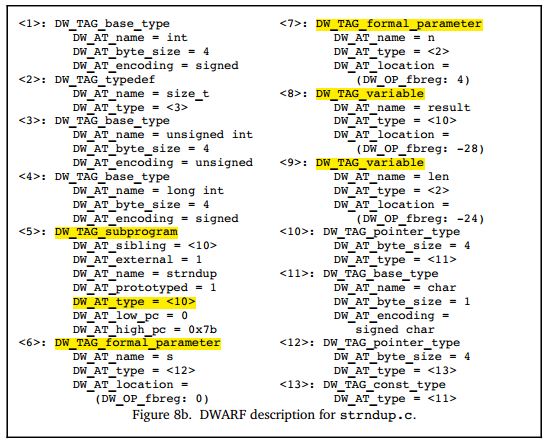
1. lexical block

Most programming language support lexical block, there’s may be some lexcical blocks in subprogram, which can be described DIE DW\_TAG\_lexcical\_block. Lexical block may contain variable and lexical block DIEs, too.

Following is an example showing how to describe a C function.



Generated Dwarf information is as following:



Referring to 1)~5) content, this example easy to be understood.

#### Compilation Unit

Most Program contain more than one source file. When building program, each source file are treated as a independent compilation unit, which will be independently compiled to \*.o (such as C), then these object files will be linked with system specific startup code and system libraries to build the executable program.

Dwarf adopts the terminology Compilation Unit from C as the DIE name, DW\_TAG\_compilation\_unit. The DIE contains general information about the compilation, including the directory and name of the file name, the used programming language, producer that generated the Dwarf information and the offsets to help locate the line number and macro information.

If the compilation unit takes up continuous memory (i.e., it’s loaded into memory in one piece), then there’re values for the low and high memory addresses for the unit, which are low pc and high pc attributes. This helps debugger determine which compilation generate the code (instruction) at particular memory address much more easiliy.

If the compilation is not continuous, then a list of the memory address that the code takes up is provided by the compiler and linker.

#### Data Encoding

Dwarf data conceptually is a tree of DIE, DIE may has children or siblings, each DIE may has several attributes. Dwarf data is unwieldly, so it must be compressed to reduce the size, then the compressed data is stored into the object file.

Dwarf provides serveral methods to compress the data.

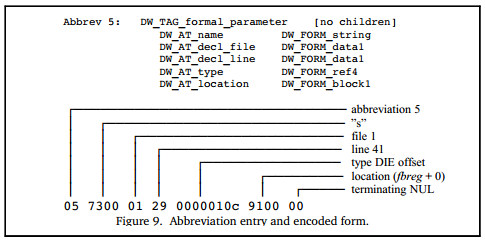
1. Use prefix traversal to flatten the tree.

Prefix traversal the Dwarf tree, the accessed tree node DIE is stored. By this way, the links between DIE and its children DIEs and sibling DIEs are eliminated. When reading Dwarf data, maybe jumping to the next sibling DIE is needed, the sibling DIE can be stored as an attribute in current DIE.

1. Use abbreviation to avoid store duplicated values.

Instead of storing the value of the TAG and attribute-value pairs, only an index into a table of abbreviations is stored, followed by attributes codes. Each abbreviation gives the TAG value, a flag indicating whether the DIE has children, and a list of attributes with the type of value it expects.

Figure 9 is an example of using abbreviation:



Less commonly used are features of Dwarf 3 and 4 which allow references from one compilation unit to the Dwarf data stored in another compilation unit.

### Other Dwarf Data

#### Line Number Table

The Dwarf line number table contains the mapping between the memory address of executable code of a program and the source lines that corresponds to these address.

In the simplest form, this can be looked as a matrix with one column contains the instruction address while another column contains the source line triplet (file, line, column). When setting a breakpoint of a source line, query this table to find the first instruction and set a breakpoing. When program has a fault during execucation, query current instruction address related source line to analyze it.

As we imagined, if this table were stored with one row one each instruction, this line number table would be too huge. How to compress it? Dwarf encodes it as a sequence of instructions called a line number table program. These instructions are interpreted by a simple finite state machine to recreate the complete line number table. Also, when recreating the complete line number table, only the first machine instruction of each source statement is stored into the table.

By means of this, line number table is compressed.

#### Macro Information

Most debuggers have a very difficult time displaying and debugging code which has macros. The user sees the original source file, with the macros, while the code corresponds to whatever the macros generated.

Dwarf includes the description of the macros defined in the progam. This is quite rudimentary information, but can be used by a debugger to display the values for a macro or possibly translate the macro into the corresponding source language.

#### Call Frame Information

Every processor has a certain way of determining how to pass parameters and return values, this is defined by the processor’s ABI (Application Binary Interface). In the simplest case, all functions have the same way to pass parameters and return values, and the debuggers know exactly how to get the parameters and return values.

But actually, not every processor uses the same way to pass parameters and to return values. Besides, compilers may do some optimization to make generated instructions much smaller and faster. For example, maybe a simple function is created to use caller’s local variables as parameters instead of passing parameters to callee to avoid create frame, maybe to optimize the usage of registers, maybe others… The result is a small change takes place in the optimizations and the debugger may no longer be able to walk the stack to the calling functions.

Dwarf call frame information (CFI) provides the debugger enough information about how a function is called, how to locate the parameters to the functions, how to locate the call frame for the calling function. This information is used by the debugger to unwind the stack, locating the previous function, the location where the function is called, and the values passed.

Like the line number table, CFI is also encoded as a sequence of instructions that are interpreted to generate a table. There’s one row for each address that contains code. The first column contains the machine address, while others contain the registers’ values at when instruction at that address is executed. Like the line number table, the complete CFI is huge. Luckily, there’s very little change between two instructions, so the CFI encoding is quite compact.

#### Variable Length Data

Integer values are used throughout Dwarf to represent everything from offsets into data sections to sizes of arrays or structures. Since most values can be represented in only a few bits, this means that the data consists mostly of zeros.

Dwarf defines a variable length integer, called Little Endian Base 128 (LEB128 for signed integers or ULEB128 for unsigned integers), which compresses the bytes taken up to represent the integers.

Wiki: https://en.wikipedia.org/wiki/LEB128

#### Shrinking Dwarf data

The encoding schemes used by Dwarf significantly reduce the size of the debugging information compared to the Dwarf v1. But unfortunately, many programs’ debugging information generated by compilers still can be very large, frequently larger than the executable code and data.

Dwarf offers ways to further reduce the size of the debugging data. This part could be neglected here for now, as far as I am concerned.

#### ELF Sections

While Dwarf is defined in a way that allows it to be used with any object file format, it’s most often used with ELF.

Each of the different kinds of Dwarf data are stored in their own section. The names of these sections all start with prefix ‘.debug\_’. For improved efficiency, most references to Dwarf data use an offset from the start of the data for current compilation. This avoids the need to relocate the debugging data, which speeds up program loading and debugging.

The ELF sections and their contents are:

1. .debug\_abbrev, abbreviations used in .debug\_info
2. .debug\_arranges, a mapping between memory address and compilation
3. .debug\_frame, call frame info
4. .debug\_info, the core Dwarf data containing DIE
5. .debug\_line, the line number program (sequence of instructions to generate the complete the line number table)
6. .debug\_loc, location descriptions
7. .debug\_macinfo, macro information
8. .debug\_pubnames, a lookup table for global object and functions
9. .debug\_pubtypes, a lookup table for global types
10. .debug\_ranges, address ranges referenced by DIEs
11. .debug\_str, string table used by .debug\_info
12. .debug\_types, type descriptions

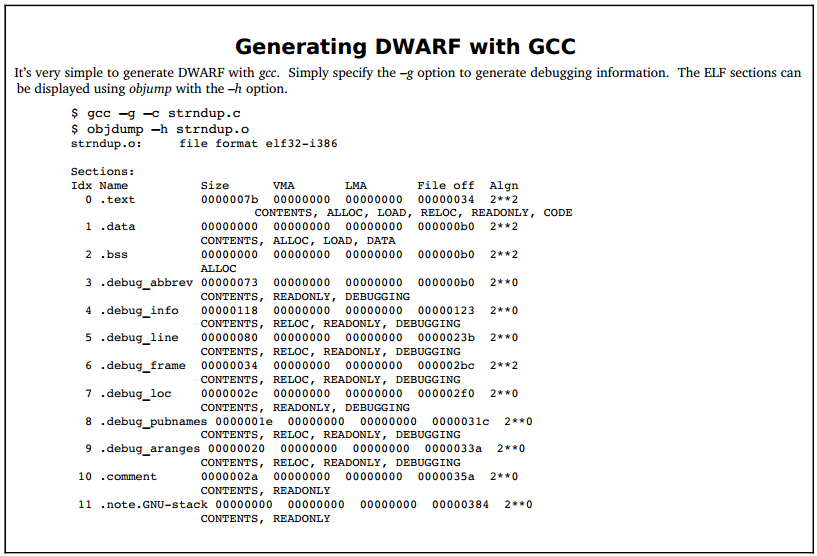
### Summary

The basic concepts for the Dwarf are quit straight-forward. A program is described as a tree with nodes representing the various functions, data and types in the source in a compact language and machine-independent fashion.

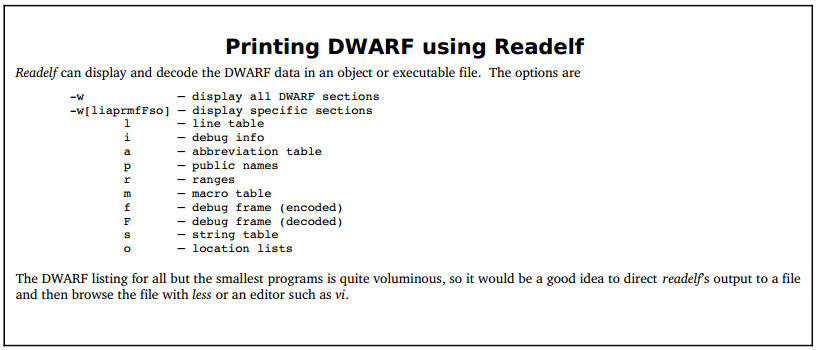
The line table provides the mapping between the executable instructions and the source that generated them. The CFI describes how to unwind the stack.

There is quite a bit of subtlety in Dwarf as well, given that it needs to express the many different nuances for a wide range of programming languages and different machine architectures.

By using ‘**gcc -g -c filename.c**’ can generate the Dwarf debugging information and stored it into the object file filename.o.



By using ‘**readelf -w**’ can read and display all Dwarf debugging information. By using ‘readelf -w’ and specifying flags can read and display specific Dwarf debugging information.



## What to get

## How to get

# Golang Debugger

## Basic

### StartProcess

func Command(name string, arg ...string) \*Cmd

func FindProcess(pid int) (\*Process, error)

### TraceProcess

1. func PtraceAttach(pid int) (err error)

According to Linux manual of ptrace, a ptrace attach syscall can only trace a single physical thread. If the debugged process has multiple threads to trace, each tracee thread must be traced by a ptrace call.

Also, the tracer thread must be locked to the same physical thread, pay attention to this if you want to use golang to develop a debugger, because goroutine maybe offered to another physical thread to schedule

.

1. func (p \*Process) Wait()

After tracee thread stopped or exited, OS will send a signal to the tracer to notify the tracee’s state, then the tracer thread can use ptrace syscall to inspect the tracee’s memory and registers.

### Inspect Registers

1. func PtraceGetRegs(pid int, regsout \*PtraceRegs) (err error)

This function can be used to get tracee’s registers’ data.

1. func PtraceSetRegs(pid int, regs \*PtraceRegs) (err error)

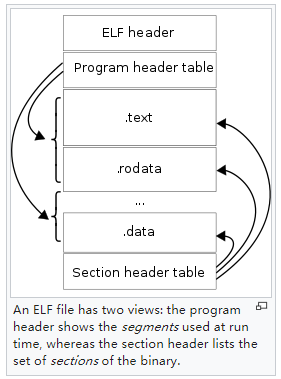
This function is often used to update Program Counter (PC) when handle breakpoints.

### Symbol Table, LinenumTable

#### debug/elf

ELF, short for Executable and Linkable Format, is a common standard file format for executable files, object code, shared libraries, and core dumps. It is commonly used in Unix and Linux.

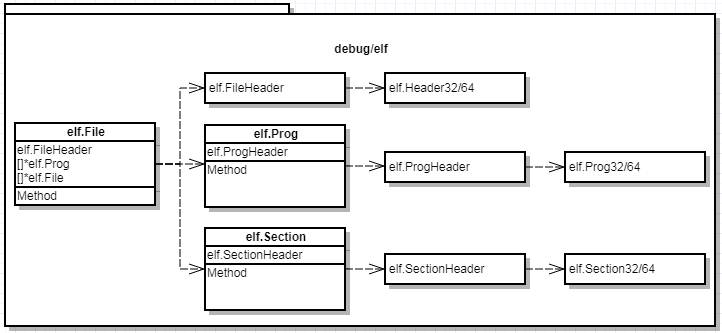
ELF format is as following:



Debug/elf, this package provides a way to read the ELF File Header, Program Header Table, Section Header Table of elf file.

1. elf.go, defines the constants and datatype for primitive ELF File Header, Program Header Table, Section Header Table, etc.
2. file.go, beyond of elf.go, defines the File, File Header, Prog, ProgHeader, Section, SectionHeader, etc.

These datatype’s relation is as following:



#### debug/gosym

debug/gosym, this package provides a way to build Symbol Table and LineTable, etc.

1. pclintab.go, it builds the line table *gosym.LineTable*, which handles LineToPC, PCToLine, etc.
2. symtab.go, it builds the symbol table *gosym.Table*, which handles LookupSym, LookupFunc, SymByAddr, etc. Through the lookuped symbol, we can also retrieve some important information, such as retrieving line table from a Func.

#### debug/dwarf

Dwarf v2 aims to solve how to represent the debugging information of all programming languages, there’s too much to introduce it. Dwarf debugging information may be generated and stored into many debug sections, but in package debug/dwarf, only the following debug sections are handled:

1. .debug\_abbrev
2. .debug\_info
3. .debug\_str
4. .debug\_line
5. .debug\_ranges
6. .debug\_types
7. const.go, it defines the constansts defined in Dwarf, including constants for tags, attributes, operation, etc.
8. entry.go, it defines a DIE parser, type *dwarf.Entry* abstracts a DIE entry including 3 important members, Tag(uint32), Field{Attr,Val,Class}, Children(bool).

It defines a DIE Reader for traversing the .debug\_info which is constructed as a DIE tree via:

*f, e := elf.Open(elf)*

dbg, e := f.DWARF()

r := dbg.Reader()

*for {*

*entry, err := r.Next()*

*if err != nil || entry == nil {*

*break*

*}*

*//do something with this DIE*

*//…*

*}*

1. line.go, each single compilation unit has a .debug\_line section, it contains a sequence of LineEntry structures. In line.go, a LineReader is defined for reading this sequence of LineEntry structures.

*func (d \*dwarf.Data) LineReader(cu \*Entry) (\*LineReader, error)*, the argument must be a DIE entry with tag TagCompileUnit, i.e., we can only get the LineReader from the DIE of compilation unit.

*f, e := elf.Open(elf)*

*dbg, e := f.DWARF()*

*r := dbg.Reader()*

*for {*

*entry, \_ := r.Next()*

*if err != nil || entry == nil {*

*break;*

*}*

*// read the line table of this DIE*

*lr, \_ := dbg.LineReader(entry)*

*if lr != nil {*

*le := dwarf.LineEntry{}*

*for {*

*e := lr.Next(&le)*

*if e == io.EOF {*

*break;*

*}*

*}*

*}*

*}*

1. type.go, Dwarf type information structures.
2. typeunit.go, parse the type units stored in a Dwarf v4 .debug\_types section, each type unit defines a single primary type and an 8-byte signature. Other sections may then use formRefSig8 to refer to the type.
3. unit.go, Dwarf debug info is split into a sequence of compilation units, each unit has its own abbreviation table and address size.

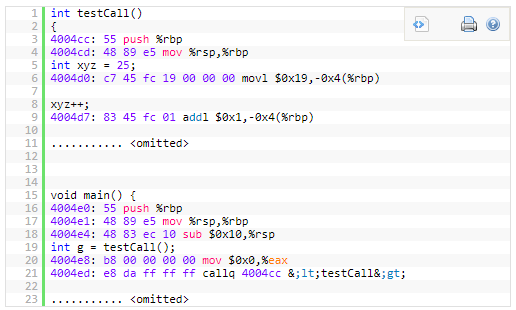
### Inspect Call Frame

#### Purpose

Debuggers often need to view and modify the state of frame on the call frame stack. To be able to view or modify a frame that is not on the top of the call frame stack, the debugger must *virtually unwind* the stack of call frame util it finds the frame of interest. Virtually Unwind means it has the logic effect of returning from the current subroutine to its predecessor, but it can still *rewind* the stack back to the state it was in before it attempted to unwind it, just thinking *gdb commands like bt, frame $n, info frame*.

Dwarf v2 provides a way to describe the call frame information which is stored in section .debug\_frame, and the basic idea of .debug\_frame is to allow debuggers to figure out how to unwind frames, i.e., how to restore the previous frame from any instruction executing using the current frame.

For Example, consider the following assembly (with C code inline – objdump -S):



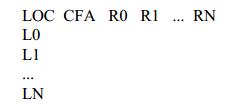
For example, when we are inside testCall (i.e. from 4004cc onward), at every instruction, the .debug\_frame information should allow us to restore all registers (such as rbp, rsp, eax etc.) to their previous status in the caller’s frame (here the caller is main). **This is done by keeping a table structure that keeps track of all the changes that happens with each register for every line of assembly.**

#### CIE & FDE

As is mentioned above, we need keeping a table structure that keeps track of all the changes that happens with each register for every line of assembly.

#### Table Structure

Dwarf supports virtual unwinding by defining an architecture independent basis for recording how procedures save and restore registers throughout their lifetimes. The basis must be augmented on some machines with specific information that is defined by either an architecture specific ABI authoring committee, a hardware vendor, or a compiler producer. The body defining a specific augmentation is referred to below as the augmenter.



In this table:

1. LOC, it’s the address of assembly (for example 4004cc), possible values are the address of each instruction.
2. CFA, it’s the call frame address, which uses a register and an offset to calculate the CFA.
3. R0~RN: they’re the registers that the particular architecture has, they record whether the registers’ values has been changed and how the value is changed.

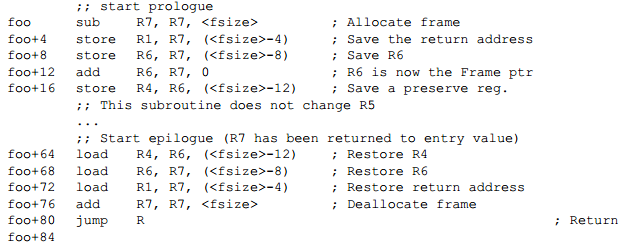
Possible values are:

* Undefined(u): has no value in previous frame, it’s not preserved by a callee.
* Same(s): hasn’t been modified from previous frame, it’s not preserved by the callee, but the callee doesn’t modify it.
* Offset(N): previous value of this register is stored at the address CFA+N where CFA is the current CFA value and N is a signed offset.
* Register(R): previous value of this register is stored at the register number R.
* Architectural: the rule is defined externally to this specification by the augmenter.

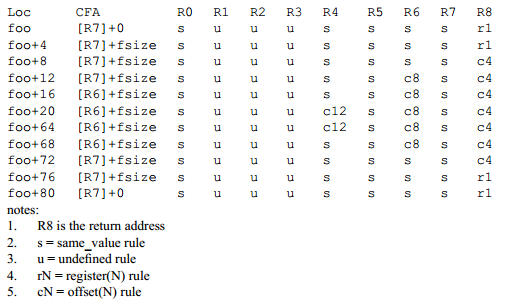
In essence, the .debug\_frame section and its corresponding cryptic tags are used to help the user construct such table.

#### Build Table

Take following code as an example:

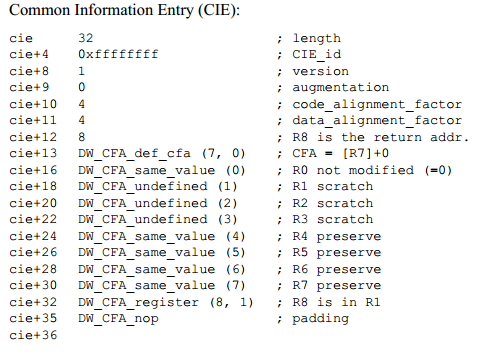


The .debug\_frame can be used to generate following table:



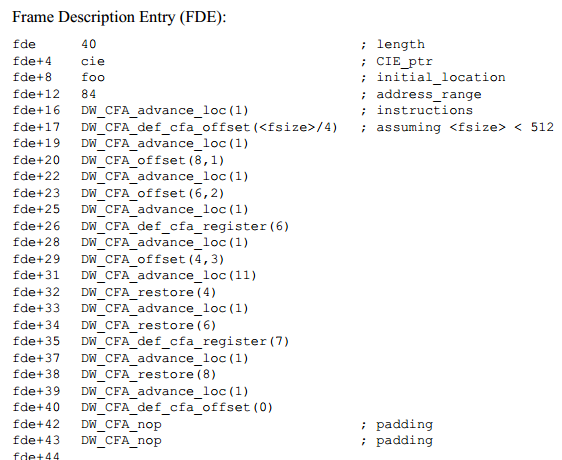
How to build this table?

1. In section .debug\_frame, each compilation unit has a single Common Information Entry (CIE), CIE describes how to built the first row in this table by the initial instructions included in this CIE, for example, *cie+13~cie+35* can be used for building the first row *foo [R7]+0 s u u u s s s s r1*.



*A CIE holds information that is shared among many Frame Descriptions. There is at least one CIE in every non-empty .debug\_frame section. Often, a compilation unit has a single CIE entry, which may contains a slice of FDE entries. A CIE has the following fields: length, CIE\_id, version, augmentation, code\_alignment\_factor, data\_alignment\_factor, return\_address\_register, initial\_instructions, padding.*

1. Each CIE has a slice of FDEs, among which each FDE has a pointer to this CIE. These FDEs can be used for building the following rows in this table by the included instructions in this FDE.



*A FDE holds information about a call frame, it has the following fields: length, CIE\_pointer, initial\_location, address\_range, instructions.*

*The field* ***initial\_location*** *refers to the address of the first assembly in this FDE, the* ***address\_range*** *refers to the number of bytes of Dwarf DW\_CFA\_... instructions used for recreating the table row. When specifying a PC, we can use this expression* ***(PC - fde.initial\_location < fde.address\_range)*** *to check whether this PC (instruction address) is within the range of this FDE. Actually we need to traverse the FDEs in the CIE to check which FDE covers the PC.*

Ref: <http://ucla.jamesyxu.com/?p=231>

1. Call Frame Instructions

* DW\_CFA\_advance\_loc(delta), advance delta \* code\_alignment\_factor,
* DW\_CFA\_def\_cfa\_register(R), takes a single unsigned LEB128 argument R representing a register number. The required action is to define the current CFA rule to use the provided register (but to keep the old offset).
* DW\_CFA\_def\_cfa\_offset(N), takes a single unsigned LEB128 argument N representing an offset. The required action is to define the current CFA rule to use the provided offset (but to keep the old register).
* DW\_CFA\_offset(R, N), takes R as register number and N as offset to CFA, this create a offset(N) rule on register numbered R.
* DW\_CFA\_restore(R), takes R as register number, it applies the rule assigned by CIE to R before to this register again in this new row.

According to this instructions stored in CIE or FDEs, we can know how to build the table structure.

#### Calc ReturnAddress

How to calculate the ReturnAddress given any instruction in current frame?

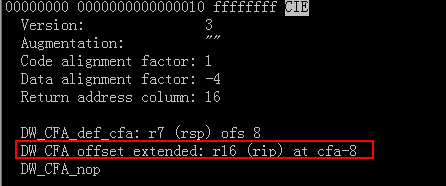
1. Calculate the CFA according to the table mentioned in 3.1.5.4.

CFA is specified by a register (golang uses RSP) and an offset, because Dwarf instruction DW\_CFA\_... cannot affect RSP’s value, so when PC of instruction is specified, we can directly execute instructions in the single FDE covering this PC to get the CFA.

*Remark: If we want get other registers’ values, we must execute all the predessesor FDEs’ instructions, then excute the FDE’s instructions which covering this PC. Only by this, other registers’ values can be affected rightly.*

1. Confirm which register holds the return address by CIE.return\_address\_register, this register maybe a virtual one (not an actual machine register), such as CFA-8 used in golang’s CIE, which refers to a memory address, then we can use ptrace to peek the data stored in this memory address, this is the return address.

Here is an golang CIE sample, DW\_CFA\_offset\_extended: r16 (rip) at cfa-8, it defines that register numbered 16 is not an actual machine register, actually it points to a memory unit with address cfa-8.



*Note:*

*CFA is short for Canonical Frame Address or Caller Frame Address rather than ReturnAddress. Here’s an issue which describes this point:* [*https://github.com/derekparker/delve/issues/1181*](https://github.com/derekparker/delve/issues/1181)*.*

Also, you can checkout this repo to validate this issue: <https://github.com/hitzhangjie/golang-debugger> at revision <9097961>.