OpenDTrace Specification

Version 0.1 - DRAFT

Robert N. M. Watson, George Neville-Neil, and Domagoj Stolfa

BAE Systems, The University of Cambridge,
and Memorial University Newfoundland

August 16, 2017

Approved for public release; distribution is unlimited. Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8650-15-C-7558 ("CADETS") as part of the DARPA Transparent Computing research program. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

Abstract

OpenDTrace is a dynamic tracing facility offering full-system instrumentation, a high degree of flexibility, and portable semantics across a range of operating systems. Originally designed and implemented by Sun Microsystems (now Oracle), user-facing aspects of OpenDTrace, such as the D language and command-line tools, are well defined and documented. However, OpenDTrace's internal formats – the DTrace Intermediate Format (DIF) and DTrace Object Format (DOF) – have primarily been documented through source code comments rather than a structured specification. This technical report specifies these formats in order to better support the development of more comprehensive tests, new underlying execution substrates (such as just-in-time compilation), and future extensions. Throughout this report we use the name `OpenDTrace` to refer to the open source project but retain the name `DTrace` when referring to data structures such as the DTrace Intermediate Format. OpenDTrace builds upon the foundations of the original DTrace code but provides new features, which were not present in the original. This document acts as a single source of truth for the current state of OpenDTrace as it is currently implemented and deployed.

Acknowledgments

The authors of this report thank the creators of DTrace for a spectacular contribution to the field of operating-system design, and in particular for designing the data structures, instructions, and other elements of DTrace described in this specificaiton. Some of the text in this specification has been excerpted from the excellent comments present in the original source code.

XXXRW: Ideally, there would be a list of DTrace folk who we thanked for their reviewing this document, here – e.g., Brendan, Brian, and others.

The authors of this report also thank other members of the CADETS team, and our past and current research collaborators at BAE Systems, the University of Cambridge, and Memorial University Newfoundland:

Jon Anderson	David Chisnall	Brooks Davis	Khilan Gudka
Graeme Jenkinson	Ben Laurie	Ilias Marinos	Peter G. Neumann
Greg Sullivan	Rip Sohan	Amanda Strnad	Arun Thomas
Bjoern Zeeb			

Finally, we are grateful to Angelos Keromytis, DARPA Transparent Computing program manager, who has offered both technical insight and support throughout this work.

Contents

Chapter 1

Introduction

OpenDTrace is a dynamic tracing facility integrated into the Solaris, FreeBSD, and Mac OS X operating systems – with ports also available for Linux and Windows. Dynamic tracing allows system administrators and software developers to develop short scripts (in the D programming language) that instruct OpenDTrace to instrument aspects of system operation, gather data, and present it for human interpretation or mechanical processing. While there is excellent documentation available for the D programming language, command-line tools, and OpenDTrace-based investigation and operation, the internal formats to OpenDTrace are generally documented via the source code. This report acts as a de facto specification for those formats, including the DTrace Intermediate Format (DIF), which is a bytecode that D scripts are compiled into for safe execution within the kernel, and the DTrace Object Format (DOF), which bundles together complete scripts along with their associated constants and metadata.

1.1 Background

XXRW: Ideally, there would be a more detailed description of both the usage model, and also the architectural elements, of DTrace here. That and some citations to DTrace documentation, the FreeBSD/Solaris books, etc.

The original DTrace code was designed and developed by Sun Microsystems to solve a particular problem, being able to instrument systems that were currently deployed, without requiring the recompilation of any code.[2]. The DTrace system was written in a portable style typical of code from the Sun Microsystems Kernel Development group in the early 2000s. Shortly after the release of the original DTrace system a port was made, by John Birrel, to the FreeBSD Operating System. A port was also made by Apple Computer to their Mac OS X at about the same time. DTrace gained popularity as a dynamic tracing system throughout the first decade of the 21st Century and its usage is well documented.

OpenDTrace is made up of several components, including kernel code, user space libraries, and command line tools. The OpenDTrace system uses information generated during code compilation to expose a set of trace points with which users and programs can interact. These trace points can be the entry and exit points of functions as well as system calls, or they can be arbitrary points in the instruction stream, marked out with a set of standardized macros. From the user's point of view tracing is activated by a command line program, `dtrace`, but any program that is compiled with the OpenDTrace libraries may initiate tracing, so long as it has sufficient privileges.

The OpenDTrace privilege model is relatively simple, any program that wishes to trace another program must be running with *root* privileges. Some operating systems, such as Illumos, provide a more nuanced privilege model, the details of which are discussed further in Section 2.2.

Tracepoints are collected into one of many *providers* which dictate the capabilites of the tracepoint and how it interacts with the overall tracing system. Providers exist for system calls (`syscall`), function boundary tracing (`fbt`), timing services (`profile`), as well as specific subsystems such as the network (`ip`, `tcp`), filesystem (`vfs`) and process scheduler (`proc`). Arbitrary trace points can be added to the kernel via the statically defined trace point (`sdt`) provider. User space programs are traced either with the `pid` provider or using the statically defined trace point (`usdt`) provider.

1.2 Version History

0.1 This is the first version of the *OpenDTrace Formats Specification*, made available for early review and collaborative development.

1.3 Document Structure

This report specifies a number of aspects of OpenDTrace's operation:

The OpenDTrace Operational Model described in Chapter 2 gives a general overview of the internals of the OpenDTrace system, including the privilege model, trace-point format and a description of how user space programs are traced.

The D Langauge described in Chaper 3 provides a full description of the D language, which is the domain specific scripting language used to create more complex data queries and to perform data reduction after tracepoint data has been captured.

The Compact Trace Format described in Chapter 4 explains the data extracted from compiled object code that is used by OpenDTrace to create trace points and extract function arguments and types.

The DTrace Object Format (DOF) described in Chapter 5 is a file-like format linking together a set of sections describing OpenDTrace code, string constants, and other aspects of a complete compiled OpenDTrace script.

The DTrace Intermediate Format (DIF) is the bytecode that the executable elements of OpenDTrace scripts are compiled to. This is a simple RISC-like instruction set with constrained execution properties (e.g., only forward branches). Chapter 6 describes the instruction format and common instruction semantics.

DTrace Instructions are the individual RISC instructions performing a variety of operations including register access, memory access, arithmetic operations, and calling various built-in subroutines available to scripts in execution. Chapter 7 enumerates the instructions, their arguments, and their semantics.

DTrace Variable Records describe the set of variables (local, global, or thread-local) operated on by a OpenDTrace script. Chapter 8 specifies this record format.

DTrace Subroutines are available to scripts, providing access to higher-level behavior, such as memory copying, string comparison, and so on. Chapter 9 describes the available built-in subroutines.

Chapter 2

Operational Model

The components that make up OpenDTrace interact with each other to implement an operational model for dynamic tracing. At the highest level there are three components to OpenDTrace: tools, such as `ctfconvert` which take compiled object code and generate new ELF/DWARF sections that capture type information, the kernel module, which is responsible for adding and removing trace points at run time, and the libraries, which tie all of the components together. Users interact with OpenDTrace via the `dtrace` command line tool.

The DTrace kernel model is the heart of the DTrace framework. This module is responsible for the coordination of all other components used in instrumentation. It keeps track of all registered providers and informs them when to enable or disable their probes. When a probe fires, the OpenDTrace kernel module is responsible for executing the necessary instrumentation code and providing the data to any consumers.

The kernel module is also the intermediary between the DTrace user interface and the providers. When compiling user scripts, the kernel module provides the D compiler with probe arguments and types. Once compiled, scripts are pushed into the kernel as Enabling Code Blocks (ECBs) to be executed when probes fire. After each ECB is executed, the data is handed back to user space where the `dtrace` command line tool, or other programs linked against the OpenDtrace libraries can manipulate or display the data to end users.

Providers in OpenDTrace encapsulate the probe points that are used to instrument code and provide data to the end user. A provider defines both a set of probe points as well as the standard by which the system interacts with that set of probe points. For example the Function Boundary Tracepoint (fbt) provider defines `return` trace points in which only arguments zero (0) and one (1) are valid and which always contain the return value and return address respectively, whereas no other provider has such a definition.

OpenDTrace has a base set of providers that are shipped as part of the system, (see Table XXX), but developers are free to create their own to expose more or different information from their code. Providers can be developed either for the kernel, in which case they are defined as kernel modules, or for user space, as part of the User Defined Static Tracing (USDT) system.

A provider is simply a collection of probe points. Probe points are functions that are run when certain points in the code are reached. The probe gathers data of interest and passed data back into the OpenDTrace kernel module for further processing. Since the overhead of probes should be avoided when data is not required, the provider is responsible for tracking when probes are enabled and implementing a mechanism for the kernel module to update their state.

The user space interface to DTrace is the DTrace command line utility, `drace(1)`. The `dtrace` command line utility handles all run time interaction with the OpenDTrace system, such as submitting scripts for execution as well as configuring options as memory usage, and how often the system should flush data from the kernel. The complete syntax and set of options for the `dtrace` command is given in the manual page.

The majority of the DTrace CLI functionality is provided through calls to the DTrace user-space library, `libdtrace`, which is responsible for setting DTrace options, compiling D scripts, and passing compiled D code to the kernel for execution. `libdtrace` provides the mechanism for all interactions with DTrace in the kernel.

2.1 Probe Life Cycle

An example of instrumentation with OpenDTrace is shown in Figure 2.1. For the purposes of this example it is assumed that the DTrace kernel module has already been loaded during system boot. We ignore execution of code withing any of the providers and only discuss the interactions between components. Internal functions of interest within the kernel module and CLI are shown.

When a provider is first loaded it registers itself with the OpenDTrace kernel module (1). The registration process causes the provider to enumerate all of its available probes and the probes are also disabled by default.

The provider and kernel module remain idle until instrumentation is requested. Instrumentation is requested via the `dtrace` command in cooperation with the `libdtrace` library. The the user provides a D script, specifying the code to be run when a probe fires (2). When the `dtrace` command executes it initializes the `libdtrace` library, which in turn causes the kernel module to initialize some tracing state and set up memory buffers to stored the trace data.

The `libdtrace` library then compiles the D script (3). As part of this process the compiler queries the kernel module to determine the arguments for probes of interest via an ioctl (3a). The kernel in turn queries the provider for a description of the probe arguments which are returned to the compiler. If the arguments discovered by the kernel module do not match those supplied in the D script the compiler will signal an error and abort compilation of the D script. If the script did not supply any type information the compilation will complete and any mismatch will result in a runtime error.

The result of the D script compilation is an Enabling Code Block (ECB). The ECB is provided to the kernel module (3c) which stores it with others in a tree like structure. Once the ECB is safely stored in the kernel, the kernel module tells the provider to enable the probes that are to be instrumented.

When a code execution reaches a point that has an enabled probe, the probe fires and a call is made into the kernel module (5). The kernel module then walks through the tree of ECBs, executing any that match the probe that was fired (6). The captured data is written into the buffer created when `libdtrace` was initialized. At a later point the data is copied out of the kernel by the library (7), and then the final results are made available to the end user (8).

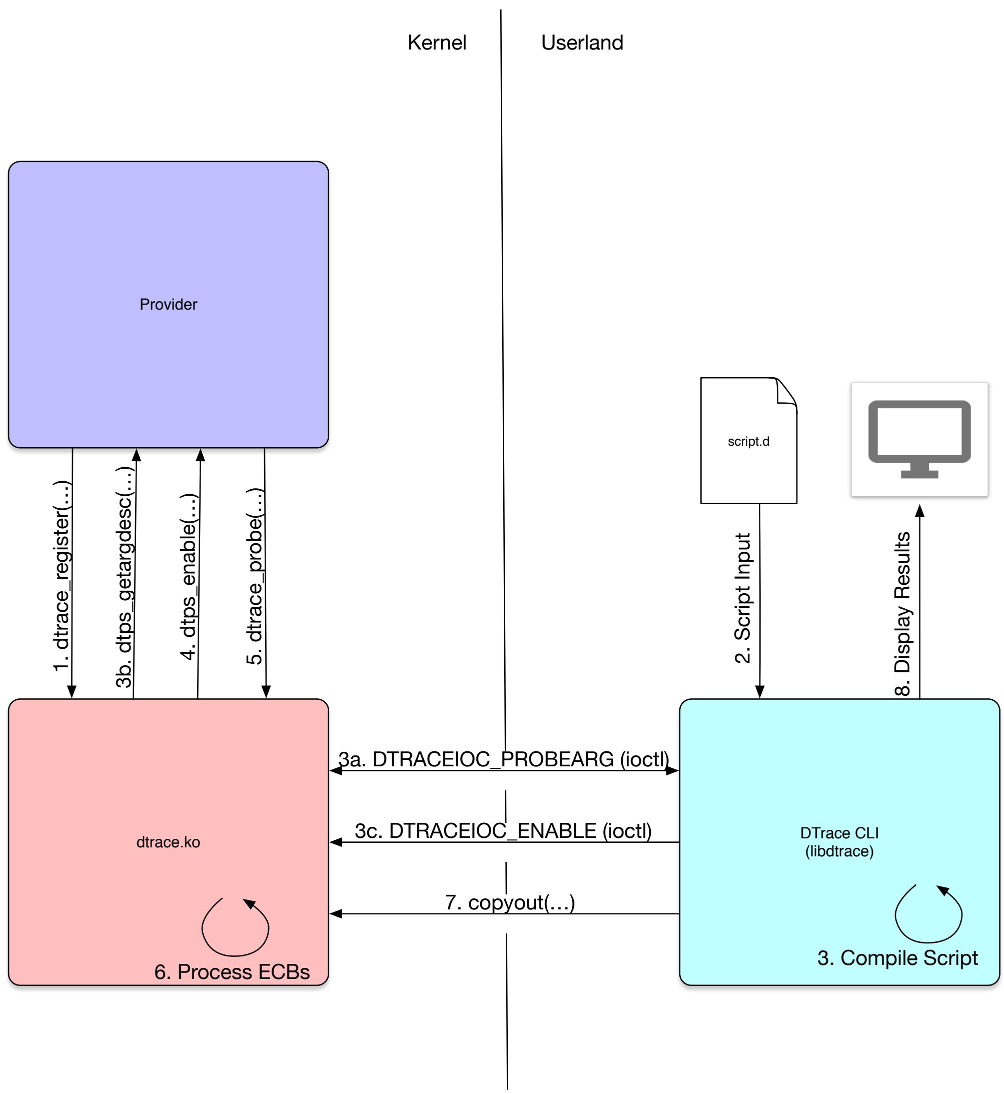

Figure 2.1: Typical lifecycle of an instrumentation using DTrace

2.2 Privilege Model

2.3 Tracepoint Format

2.4 User Space Tracing

Chapter 3

The OpenDTrace D Language

The D language is a language inspired by the AWK programming language [1] and the C programming language [3][2]. In this chapter, we give a formal definition of the D programming language that is a part of OpenDTrace, as well as elaborate on its properties in multithreaded environments.

3.1 Grammar definition

3.2 Safety

3.3 Aggregations

3.4 Variables

DTrace implements three different scopes of variables: global, thread-local and clause-local. Global variables are visible to every probe and across all threads, allowing the user to write scripts that carry state across multiple threads should it be necessary. Thread-local variables are only visible within a single software thread, they are represented in source code as prefixed with `self->`. Clause-local variables are implemented on a per-thread basis and are identified by the prefix `this->`. Clause-local variables should be initialised in each probe before their use, as the value is otherwise considered undefined.

```
dtrace:::BEGIN
{
  num_syscalls = 0;
}

syscall:::entry
{
  num_syscalls++;
}

dtrace:::END
{
  printf("Number of syscalls: %d\n", num_syscalls);
}
```

Figure 3.1: Global Variable Usage

3.4.1 Global variables

3.4.2 Thread-local variables

3.4.3 Clause-local variables

3.5 Multithreading

When tracing, DTrace guarantees that it can not be preempted inside of the `dtrace_probe` function, but it does not guarantee that everything in the executing DIF will be thread-safe. DTrace does not allow access to locking primitives, because a programming error might violate the safety guarantees that OpenDTrace was designed to provide.

3.5.1 Global variables

Global variables are not stored in thread-local storage, while thread-local and clause-local variables are. In a multithreaded environment, global variables should be used sparingly. While it is evident that a value stored in a global variable may be overwritten by another probe at any time, there is more subtle behavior at hand. Consider the following example:
Because DIF performs all of sit operations on a virtual machine's registers as opposed to variables in memory, the ++ operator is not atomic. When we compile the syscall:::entry clause, we get the following DIF output:
This DIF section is safe, as long as the num_syscalls variable is not visible from any other thread. If it is visible and accessible from another thread, it suffers from a race condition which results in wrong information being given to the user. Consider the following:

```
ldgs %r1, num_syscalls /* Load the current value into %r1 */
setx %r2, inttab[0]    /* Load 1 into %r2 */
add  %r2, %r1, %r2     /* Add %r1 and %r2 and store into %r2 */
stgs %r2, num_syscalls /* Store the result back into num_syscalls */
```

Figure 3.2: DIF Assembly

```
      Thread 1                       Thread 2
ldgs %r1, num_syscalls
                              ldgs %r3, num_syscalls
                              setx %r4, inttab[0]
                              add  %r4, %r3, %r4
setx %r2, inttab[0]
add  %r2, %r1, %r2
stgs %r2, num_syscalls
                              stgs %r4, num_syscalls
```

Figure 3.3: Race Condition

It is clear that the value in the **`r2`** register will be lost because the register **`r4`** is stored to the same location afterwards. It is worth noting that this behaviour is not observed because the thread was preempted, but simply by the fact that DTrace does not guarantee any ordering outside of each CPU core. This behaviour applies to all of the operations performed on global variables and as a result, they should only be used in probes that are guaranteed to fire on a single thread.
Often the desired behaviour with global variables can be achieved through aggregations. The above script ought to be written in the following way in order to be thread-safe:

3.5.2 Thread-local variables

As mentioned in Subsection 3.4.2, thread-local variables are only visible within a single thread.

```
    syscall:::entry
    {
      @num_syscalls = count();
    }

    dtrace:::END
    {
      printa(@num_syscalls);
    }
```

Figure 3.4: Avoiding the race condition

3.5.3 Clause-local variables

Chapter 4

Compact C Type Format (CTF)

The Compact C Type Format (CTF) encapsulates all of the information needed by OpenDTrace to understand C language types such as integers, strings, floats and structures. The goal of having another section just for C type information is to provide a compact representation of the information that usually appears in the debugging sections of object files and executables. CTF only contains data types it does not contain other debugging infromation, which allows it to be far more compact. The debugging sections on a debug build of FreeBSD in 2017 take up 78 megabytes of space, while the CTF section in the same kernel take up only 800 kilobytes.

4.1 On Disk Format

CTF data is stored in its own ELF section within an object file or executable. It is meant to be stored in a format that is both compact and which is properly aligned so that it can be accessed using the mmap(2) system call.

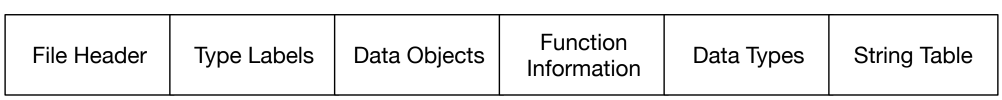

Figure 4.1: CTF Stable Storage Format

Figure 4.1 shows all of the components of the CTF section as they would be found on stable storage. The file header stores a magic number and version information, encoding flags, and the byte offset of each of the sections relative to the end of the header itself. As of this writing the most current version of CTF is version two (2). The preamble, including the magic number, version and flags, take up the first 32 bits of the header, the remaining fields take up 32 bits each, independent of the word size of the architecture.

The CTF section makes heavy use of references between the sub-sections to fully describe the datatypes in a program as well as the functions, the function's argument list, and the function's return value. The `data objects` and `functions` sections depend upon the `type` section, which encodes all of the datatypes that have been during the CTF conversion process. Each type

31 16 8 0

magic	version	flags
reference to parent label		
reference to basename of parent		
label section offset		
function section offset		
type section offset		
string section offset		
size of string section (bytes)		

Figure 4.2: Overall CTF section encoding

31 16 0

name	
info	size or type

Figure 4.3: A simple type

has a unique number and name, as well as a size and encoding. Types may refer to other, more primitive types by use of a reference, e.g. a `uint32_t` will actually refer to a `unsigned int`. Types are broken up by what they represent, referred to as their *kind*.

Table4.1 lists the kinds of base data types that are encoded by CTF. Complex data types, such as structures, are also contained in the `types` section, and are encoded as a structure with a name that references the string table.

A simple type, one who's size is less than 64 Kbytes, is stored in a `ctf_stype`, Figure 4.3. The `name` is a reference to a string in the string table. The `info` field is encoded differently for each type, as will be explained fully in the rest of this chapter. The last field is either the size, in bytes, of the structure or it is a reference to another type, encoded using the referenced type's ID. The majority of types in a C program will fit within a `ctf_stype`.

Types that are larger than 64Kbytes are encoded using a `ctf_type` structure, shown in Figure 4.4. The `name` and `info` fields of this, larger, `ctf_type` are the same as the smaller `ctf_stype`, but the `size` field is always set to `CTF_LSIZE_SENT`, the sentinal value that tells the consumer that this is a larger structure. A `ctf_type` structure can encode an extremely large type, since it provides 64 bits for the size, and that size is expressed in bytes.

The `info` field, shown in Figure4.5, is further broken down into a number of sub-fields which encoded the `kind`, `vlen` (variable length) and whether or not this is a root type `isroot`.

Each of the integral types, such as integers, floats, pointers, arrays, etc. has its own encoding. Integers are the simplest type and are unsigned by default. An integer type is encoded in a single,

`CTF_K_UNKNOWN`	unknown type (used for padding)
`CTF_K_INTEGER`	variant data is `CTF_INT_DATA()` (see below)
`CTF_K_FLOAT`	variant data is `CTF_FP_DATA()` (see below)
`CTF_K_POINTER`	`ctt_type` is referenced type
`CTF_K_ARRAY`	variant data is single `ctf_array_t`
`CTF_K_FUNCTION`	`ctt_type` is return type variant data is list of argument types (`ushort_t`'s)
`CTF_K_STRUCT`	variant data is list of `ctf_member_t`'s
`CTF_K_UNION`	variant data is list of `ctf_member_t`'s
`CTF_K_ENUM`	variant data is list of `ctf_enum_t`'s
`CTF_K_FORWARD`	no additional data; `ctt_name` is tag
`CTF_K_TYPEDEF`	`ctt_type` is referenced type
`CTF_K_VOLATILE`	`ctt_type` is base type
`CTF_K_CONST`	`ctt_type` is base type
`CTF_K_RESTRICT`	`ctt_type` is base type

Table 4.1: Kinds of CTF Base Types

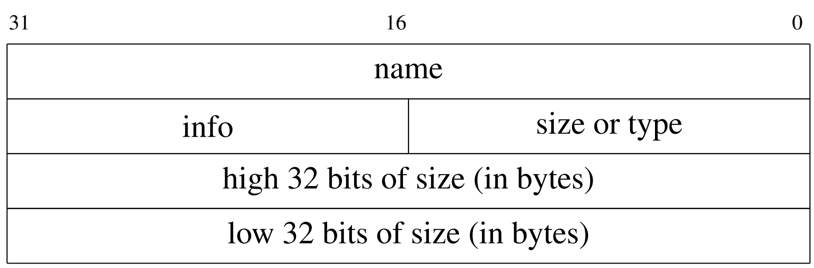

Figure 4.4: A large type

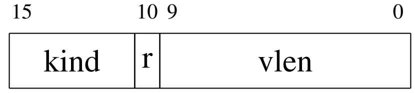

Figure 4.5: Info field encoding

31	24	16	0
flags	offset	size in bits	

Figure 4.6: Integral type encoding

`CTF_FP_SINGLE`	IEEE 32-bit float encoding
`CTF_FP_DOUBLE`	IEEE 64-bit float encoding
`CTF_FP_CPLX`	Complex encoding
`CTF_FP_DCPLX`	Double complex encoding
`CTF_FP_LDCPLX`	Long double complex encoding
`CTF_FP_LDOUBLE`	Long double encoding
`CTF_FP_INTRVL`	Interval (2x32-bit) encoding
`CTF_FP_DINTRVL`	Double interval (2x64-bit) encoding
`CTF_FP_LDINTRVL`	Long double interval (2x128-bit) encoding
`CTF_FP_IMAGRY`	Imaginary (32-bit) encoding
`CTF_FP_DIMAGRY`	Long imaginary (64-bit) encoding
`CTF_FP_LDIMAGRY`	Long long imaginary (128-bit) encoding

Table 4.2: Floating Point Encodings for CTF

32 bit, field, as seen in Figure 4.6.

The `flags` field indicates whether the integer is signed, contains character data, is a boolean or is to be displayed with a varags style of formatting.

Floating point numbers have the exact same fields to describe them but a larger number of possible flags, to match the larger number of ways in which floating point numbers may be stored. The flags and descriptions of the currently supported floating point encodings are given in Table 4.1.

The functions section encodes the function name, as well as its arguments and return value. The types of the arguments and the return value reference the `types` section. The arguments to the function are encoded as a list.

All strings are encoded in the `string table` and are referenced by a numeric id from the other sections.

Chapter 5

OpenDTrace Object Format (DOF)

5.1 Introduction

OpenDTrace programs are persistently encoded in the DOF format so that they may be embedded in other programs (for example, in an ELF file) or in the DTrace driver configuration file for use in anonymous tracing. The DOF format is versioned and extensible so that it can be revised and so that internal data structures can be modified or extended compatibly. All DOF structures use fixed-size types, so the 32-bit and 64-bit representations are identical and consumers can use either data model transparently.

5.1.1 Stable Storage Format

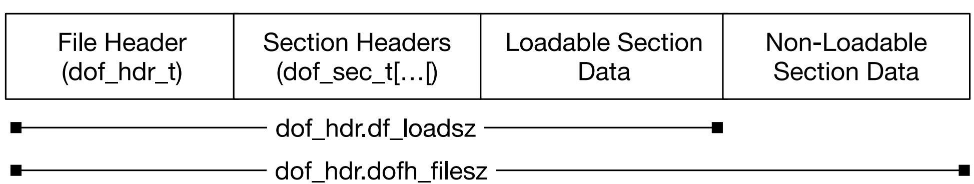

Figure 5.1: Stable Storage Format

When a DOF file resides on stable storage it is stored in the format shown in Figure 5.1. The file header stores meta-data including a magic number, data model for the instrumentation, data encoding, and properties of the DIF code within. The header describes its own size and the size of the section headers. By convention, an array of section headers follows the file header, and then the data for all loadable sections and sections which cannot be loaded, also called unloadable sections. This data layout permits consumer code to easily download the headers and all loadable data into the DTrace driver in one contiguous chunk, omitting other extraneous sections.

The section headers describe the size, offset, alignment, and section type for each section. Sections are described using a set of `#defines` that tell the consumer what kind of data is expected.

Sections can contain links to other sections by storing a `dof_secidx_t`, an index into the section header array, inside of the section data structures. The section header includes an entry size so that sections with data arrays can grow their structures.

The DOF data itself can contain many snippets of DIF (i.e. more than one DIF object or DIFO), which are represented themselves as a collection of related DOF sections. This permits us to change the set of sections associated with a DIFO over time, and also permits us to encode DIFOs that contain different sets of sections. When a DOF section wants to refer to a DIFO, it stores the `dof_secidx_t` of a section of type `DOF_SECT_DIFOHDR`. This section's data is then an array of `dof_secidx_t`'s which in turn denote the sections associated with this DIFO.

This loose coupling of the file structure (header and sections) to the structure of the DTrace program itself (enebled code block descriptions, action descriptions, and DIFOs) permits activities such as relocation processing to occur in a single pass without having to understand D program structure.

Finally, strings are always stored in ELF-style string tables along with a string table section index and string table offset. Therefore strings in DOF are always arbitrary-length and not bound to the current implementation.

Name	Loadable	Comment
DOF_SECT_NONE	N	null section
DOF_SECT_COMMENTS	N	compiler comments
DOF_SECT_SOURCE	N	D program source code
DOF_SECT_ECBDESC	Y	dof_ecbdesc_t
DOF_SECT_PROBEDESC	Y	dof_probedesc_t
DOF_SECT_ACTDESC	Y	dof_actdesc_t array
DOF_SECT_DIFOHDR	Y	dof_difohdr_t (variable length)
DOF_SECT_DIF	Y	uint32_t array of byte code
DOF_SECT_STRTAB	Y	string table
DOF_SECT_VARTAB	Y	dtrace_difv_t array
DOF_SECT_RELTAB	Y	dof_relodesc_t array
DOF_SECT_TYPTAB	Y	dtrace_diftype_t array
DOF_SECT_URELHDR	Y	dof_relohdr_t (user relocations)
DOF_SECT_KRELHDR	Y	dof_relohdr_t (kernel relocations)
DOF_SECT_OPTDESC	Y	dof_optdesc_t array
DOF_SECT_PROVIDER	Y	dof_provider_t
DOF_SECT_PROBES	Y	dof_probe_t array
DOF_SECT_PRARGS	Y	uint8_t array (probe arg mappings)
DOF_SECT_PROFFS	Y	uint32_t array (probe arg offsets)
DOF_SECT_INTTAB	Y	uint64_t array
DOF_SECT_UTSNAME		struct utsname
DOF_SECT_XLTAB	Y	dof_xlref_t array
DOF_SECT_XLMEMBERS	Y	dof_xlmember_t array
DOF_SECT_XLIMPORT	Y	dof_xlator_t
DOF_SECT_XLEXPORT	Y	dof_xlator_t
DOF_SECT_PREXPORT	Y	dof_secidx_t array (exported objs)
DOF_SECT_PRENOFFS	Y	uint32_t array (enabled offsets)

Table 5.1: DOF Section Descriptions

Chapter 6

OpenDTrace Intermediate Format (DIF)

6.1 DTrace Instruction Version

This specification describes the DTrace Intermediate Format version 2, as shipped in Illumos 5, FreeBSD 8-12, and Mac OS X 10.5-10.13

6.2 The DIF Interpreter

The DTrace Intermediate Format (DIF) interpreter executes instructions on behalf of D scripts that are associated with predicates and actions. DIF is a simple, RISC-like, instruction set where each instruction consists of a 32-bit, native-endian integer whose most significant 8 bits contain an opcode allowing the remainder of the instruction to be decoded. Interpretation is executed in a loop within the `dtrace_dif_emulate` function, which has, as its first argument, a pointer to a DIF Object or `dtrace_difo_t`. Each of the DIF objects gets executed in its own separate environment and must return a value using the `ret` instruction. Instructions are executed one at a time, until they are exhausted or an error causes interpretation to end. The DIF objects are verified in the `dtrace_difo_validate` function and the DIF interpreter ignores any bounds checking within the `dtrace_dif_emulate` function precisely because `dtrace_difo_validate` performs the necessary checks.

The following chapter describes the overall implementation of the DIF interpreter as well as how the various instructions are implemented, along with various implementation details.

6.2.1 Registers

The DTrace virtual machine is made up of eight (8) integer registers and eight (8) tuple registers. The 0th integer register always contains the value zero (0). All operations are carried out using registers **`r1`** and **`r2`** as operands and **`rd`** as the destination for all results. A comprehensive description of OpenDTrace's instructions are given in Chapter 7 and a full list and description of the built-in subroutines are given in Chapter 9.

Variable	Meaning
cc_r	Value of $r1 - r2$
cc_n	Comparison result is negative.
cc_z	Comparison result is 0.
cc_v	Overflow occurred.
cc_c	Is $r1 < r2$?

Table 6.1: Mathematical Operation Result Bits

6.2.2 Math Instructions

Instructions for mathematical operations in DIF have no concept of over or underflow. The division instructions set a flag to indicate a division by zero error.

6.2.3 Comparison and Test Instructions

DIF has three instructions, `cmp`, `scmp` and `tst` which can set various result flags, shown in Table 6.2.3. The result flags are are later used by the branch instructions to determine how to . The result flags are never returned directly to the calling DIF program but are only used internally by the interpretation routine.

XXXRW: Something about instructions

6.2.4 Branching Instructions

DIF has eleven branch instructions split into two types: signed and unsigned. The signed branching instructions take into account that the number may be negative, while the unsigned instructions are meant to be used with exclusively positive numbers. One thing all of the branching instructions have in common is that they load the new `%pc` register from the **`label`** field in the Branch Format described in Subsection 6.3.2.

6.2.5 Subroutine Calls

DTrace provides an extensive set of subroutines for use by D programs. The subroutines are implemented within the kernel code via a set of functions which are centrally dispatched from the `dtrace_dif_subr` function. Within DIF subroutines are triggered via the `CALL` instruction. The arguments to these subroutines are passed through the `tupregs` variable through use of the `PUSHTR` and `PUSHTV` instructions. The return values of the subroutines are provided through the **`rd`** register. The subroutine identifier is placed in the **`idx`** field of the W-Format described in Subsection 6.3.3. Any subroutine that is provided to DTrace *must* go via these mechanisms.

> Notice that we don't bother validating the proper number of arguments or their types in the tuple stack. This isn't needed because all argument interpretation is safe because of our load safety – the worst that can happen is that a bogus program can obtain bogus results.

According to a comment in the code, see Figure 6.2.5, argument checks are not carried out when a subroutine is called. These checks are performed in the `dtrace_difo_validate` function at load time.

XXXDS: Are these comments related to the Subroutine Calls section, or overall?
XXXRW: Something about DIF registers, `NREGS`
XXXRW: Something about the zero register
XXXRW: Something about implied status bits
XXXRW: Something about scratch memory (alloca, user copyin)
XXXRW: Something about the tuple stack
XXXRW: Something about kernel memory access
XXXRW: Something about user memory access
XXXRW: Something about "by reference"
XXXRW: Something about the integer table
XXXRW: Something about the string table

6.2.6 Local Variables

Local variables are local to the D program clause. In a D program these variables are referenced using the `this->` syntax.

6.2.7 Thread Local Variables

Thread local variables are usable in multiple D program clauses. In a D program the thread local variables are referenced using the `self->` syntax.

6.2.8 Global Variables

Global variables are global to all the clauses in a D script and are references to simple names within the D script. Space for global variables is statically allocated on each invocation of a script. Additionally, global variables are identified using the modified register format as described in Subsection 6.3.1 the case of arrays and the wide-immediate format which is described in Subsection 6.3.3 in the case of scalar variables inside of the `dtrace_dif_variable` function.

XXXRW: Something about scalars
XXXRW: Something about arrays
XXXRW: Something about aggregations
XXXRW: Something about exceptions

6.3 Instruction Format

Each instruction consists of a 32-bit, native-endian integer whose most significant 8 bits contain an opcode allowing the remainder of the instruction to be decoded. To ease parsing, three major formats (R, B, and W) are used for all DTrace instructions, capturing different types of operations:

register-to-register instructions accepting zero or more register operands; branch instructions accepting a target label as a single operand; and wide-immediate instructions that accept a 16-bit immediate used to capture both small constant values and also indices into various tables.

6.3.1 Register Format (R-Format)

This format accepts zero or more register operands, supporting instructions that include arithmetic and boolean operations, comparison and test operations, load and store operations, tuple-stack operations, and the no-op instruction.

31 24	23 16	15 8	7 0
op	r1	r2	rd

op Mandatory 8-bit operation identifier

r1, r2 Optional source registers providing input values to the operation

rd Optional destination register acting as the destination of the operation

A modified version of the Register Format is used when loading and storing data in array variables in DTrace. The main difference between the regular Register Format and the modified one used for arrays, is that the **`r1`** register location is used as the variable identificator, the **`r2`** register itself contains the optional index in the array.

31 24	23 16	15 8	7 0
op	var	r2	rd

op Mandatory 8-bit operation identifier

var The variable identifier

r2 Optional register that contains the index of the array

rd Optional destination register acting as the destination of the operation

6.3.2 Branch Format (B-Format)

This format accepts a single 24-bit integer operand identifying the label that is the branch target. It is used solely for the **BRANCH** instruction.

31 24	23 0
op	label

op Mandatory 8-bit operation identifier

label Mandatory 24-bit integer label

6.3.3 Wide-Immediate Format (W-Format)

This format accepts an 8-bit register and 16-bit integer argument (frequently an index). It is used for a range of instructions including those to load values from integer and string constant tables, as well as those that store scalar values in variables. In addition to that, it is used in the `CALL` instruction in order to specify the **`rd`** register and the subroutine identifier.

31 24	23 8	7 0
op	idx	rs|rd

op Mandatory 8-bit operation identifier

idx Mandatory 16-bit integer index

rs|rd Optional 8-bit register acting as the source or destination of the operation

Chapter 7

Instruction Reference

This chapter describes the DTrace instruction set. For a discussion of the DIF interpreter as well as an overview of how these instructions are handled see Chapter 6.

7.1 Instruction List

Name	Opcode	Description
`OR`	1	Bitwise Or
`XOR`	2	Bitwise Exclusive Or
`AND`	3	Bitwise And
`SLL`	4	Shift Left Logical
`SRL`	5	Shift Right Logical
`SUB`	6	Subtract
`ADD`	7	Add
`MUL`	8	Multiply
`SDIV`	9	Divide (Signed)
`UDIV`	10	Divide (Unsigned)
`SREM`	11	Remainder (Unsigned)
`UREM`	12	Remainder (Signed)
`NOT`	13	Bitwise Not
`MOV`	14	Move
`CMP`	15	Compare
`TST`	16	Test Equal to Zero
		See Table 7.3
`LDSB`	28	Load Byte (Signed)
`LDSH`	29	Load Halfword (Signed)
`LDSW`	30	Load Word (Signed)
`LDUB`	31	Load Byte (Unsigned)
`LDUH`	32	Load Halfword (Unsigned)
`LDUW`	33	Load Word (Unsigned)
`LDX`	34	Load Doubleword
`RET`	35	Return
`NOP`	36	No Operation
		See Table 7.4
`SCMP`	39	String Compare
`LDGA`	40	Load from Global Array
`LDGS`	41	Load from Global Scalar
`STGS`	42	Store to Global Scalar
`LDTA`	43	Load from Thread-Local Array
`LDTS`	44	Load from Thread-Local Scalar
`STTS`	45	Store to Thread-Local Scalar
`SRA`	46	Shift Right Arithmatic

Table 7.1: R-Format Instruction List (Part 1)

Name	Opcode	Description
`PUSHTR`	48	Push a reference onto the tuple stack
`PUSHTV`	49	Push a value onto the tuple stack
`POPTS`	50	Pop the tuple stack
`FLUSHTS`	51	Flush the tuple stack
		See Table 7.4
`ALLOCS`	58	Allocate scratch space
`COPYS`	59	Copy memory of requested size
`STB`	60	Store byte
`STH`	61	Store halfword
`STW`	62	Store word
`STX`	63	Store doubleword
`ULDSB`	64	Load user byte (signed)
`ULDSH`	65	Load user halfword (signed)
`ULDSW`	66	Load user word (signed)
`ULDUB`	67	Load user byte (unsigned)
`ULDUH`	68	Load user halfword (signed)
`ULDUW`	69	Load user word (signed)
`ULDX`	70	Load user doubleword
`RLDSB`	71	If accessible, load byte (signed)
`RLDSH`	72	If accessible, load halfword (signed)
`RLDSW`	73	If accessible, load word (signed)
`RLDUB`	74	If accessible, load byte (unsigned)
`RLDUH`	75	If accessible, load halfword (unsigned)
`RLDUW`	75	If accessible, load word (unsigned)
`RLDX`	77	If accessible, load doubleword

Table 7.2: R-Format Instruction List (Part 2)

Name	Opcode	Description
`BA`	17	Unconditional branch
`BE`	18	Branch if equal to zero
`BNE`	19	Branch if not equal to zero
`BG`	20	Branch if greater than (signed)
`BGU`	21	Branch if greater than (unsigned)
`BGE`	22	Branch if greater than or equal to (signed)
`BGEU`	23	Branch if greater than or equal to (unsigned)
`BL`	24	Branch if less than (signed)
`BLU`	25	Branch if less than (unsigned)
`BLE`	26	Branch if less than or equal to (signed)
`BLEU`	27	Branch if less than or equal to (unsigned)

Table 7.3: B-Format Instruction List

Name	Opcode	Description
`SETX`	37	Set register from integer table
`SETS`	38	Set register from string table
`CALL`	47	Call subroutine
`LDGAA`	52	Load from global aggregation
`LDTAA`	53	Load from thread-local aggregation
`STGAA`	54	Store to global aggregation
`STTAA`	55	Store to thread-local aggregation
`LDLS`	56	Load from local scalar
`STLS`	57	Store to local scalar
`XLATE`	78	**XXXRW: Defined but not implemented**
`XLARG`	79	**XXXRW: Defined but not implemented**

Table 7.4: W-Format Instruction List

7.2 Individual Instructions

AND: Bitwise And

Format

AND %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x03	r1	r2	rd

Description

This instruction calculates the bitwise and of the values found in registers **`r1`** and **`r2`**, placing the results in register **`rd`**.

Pseudocode

```
%rd = %r1 & %r2
```

Constraints

`r1`, **`r2`**, and **`rd`** must be less than n**`NREGS`**.

`rd` cannot be `%r0`.

Failure modes

This instruction has no run-time failure modes beyond its constraints.

OR: Bitwise Or

Format

OR %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x01	r1	r2	rd

Description

This instruction calculates the bitwise or of the values found in registers `%r1` and `%r2`, placing the results in register `%rd`.

Pseudocode

```
%rd = %r1 | %r2
```

Constraints

`r1`, **`r2`**, and **`rd`** must be less than **`NREGS`**.

`rd` cannot be `%r0`.

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SLL: Shift Left Logical

Format

SLL %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x04	r1	r2	rd

Description

This instruction shifts the value found in regiseter `%r1` left by the number of bits found in register `%r2`, placing the results in register `%rd`.

Pseudocode

```
%rd = %r1 << %r2
```

Constraints

`r1`, **`r2`**, and **`rd`** must be less than **`NREGS`**.

`rd` cannot be `%r0`.

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SRL: Shift Right Logical

Format

SRL %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x05	r1	r2	rd

Description

This instruction shifts the value found in regiseter `%r1` right by the number of bits found in register `%r2`, placing the results in register `%rd`. This instruction only operates on **unsigned** integers.

Pseudocode

```
%rd = %r1 >> %r2
```

Constraints

`r1`, **`r2`**, and **`rd`** must be less than **`NREGS`**.

`rd` cannot be `%r0`.

Failure modes

This instruction has no run-time failure modes beyond its constraints.

XOR: Bitwise Exclusive Or

Format

XOR %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x02	r1	r2	rd

Description

This instruction calculates the bitwise exclusive or of the values found in registers `%r1` and `%r2`, placing the results in register `%rd`.

Pseudocode

```
%rd = %r1 ^ %r2
```

Constraints

`r1`, **`r2`**, and **`rd`** must be less than **`NREGS`**.

`rd` cannot be `%r0`.

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SUB: subtract the value in r2 from that in r1

Format

SUB %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x06	r1	r2	rd

Description

The `sub` instruction takes the value in **`r2`** and subtracts it from that in **`r1`** placing the result in **`rd`**.

Pseudocode

```
%rd = %r1 - %r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ADD: add two values

Format

add %r1, %r2, %rd

31 24	23 16	15 8	7 0
0x07	r1	r2	rd

Description

The `add` instruction adds the the values in **`r1`** and **`r2`** and pace the results in register **`rd`**.

Pseudocode

```
%rd = %r1 + %r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

MUL: multiply two numbers

Format

MUL %rd, %r1, %r2

31 — 24	23 — 16	15 — 8	7 — 0
0x08	r1	r2	rd

Description

The `mul` instruction multiplies two numbers, contained in **`r1`** and **`r2`**, together and places the result in **`rd`**.

Pseudocode

```
%rd = %r1 * %r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SDIV: signed division

Format

SDIV %rd, %r1, %r2

31 – 24	23 – 16	15 – 8	7 – 0
0x09	r1	r2	rd

Description

The `sdiv` instruction divides the value contained in **`r2`** into that contained in **`r1`** placing the results into **`rd`**. The values in both **`r1`** and **`r2`** are first promoted to signed, 64 bit values, before the division operation is carried out.

Pseudocode

```
%rd = (int64_t)%r1 / (inst64_t)%r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

UDIV: unsigned division

Format

UDIV %rd, %r1, %r2

31 … 24	23 … 16	15 … 8	7 … 0
0x0A	r1	r2	rd

Description

The `udiv` instruction divides the value contained in **`r2`** into that contained in **`r1`** placing the results into **`rd`**.

Pseudocode

```
%rd = %r1 / %r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SREM: divide two numbers and store the remainder

Format

SREM %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x0B	r1	r2	rd

Description

The `srem` instruction divides the value contained in **`r2`** into that contained in **`r1`** placing the remainder into **`rd`**. The values in both **`r1`** and **`r2`** are first promoted to signed, 64 bit values, before the division operation is carried out. The `srem` instruction follows the remainder definition in C99 and will return a negative remainder if applicable.

Pseudocode

```
%rd = (int64_t)%r1 % (inst64_t)%r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

UREM: divide two numbers and store the remainder

Format

UREM %rd, %r1, %r2

31 – 24	23 – 16	15 – 8	7 – 0
0x0C	r1	r2	rd

Description

The `srem` instruction divides the value contained in **`r2`** into that contained in **`r1`** placing the remainder into **`rd`**.

Pseudocode

```
%rd = %r1 % %r2
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

NOT: negate a value

Format

NOT %rd, %r1

31 24	23 16	15 8	7 0
0x0C	r1	r2	rd

Description

The `not` instruction negates the value found in **`r1`** and places the result into **`rd`**.

Pseudocode

```
%rd = ~%r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

MOV: move a value

Format

MOV %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x0D	r1	r2	rd

Description

The `mov` instruction places the value found in **`r1`** into **`rd`**.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

CMP: compare two values

Format

CMP %rd, %r1, %r2

31 – 24	23 – 16	15 – 8	7 – 0
0x0E	r1	r2	rd

Description

The `cmp` instruction compares the values in **`r1`** and **`r2`**, via subtraction, and sets the various comparison bits based on the results. The comparison bits, shown in Table6.2.3, are used by the branch instructions to make decisions about where the program will execute next.

Pseudocode

```
cc_r = %r1 - %r2;
cc_n = cc_r < 0;
cc_z = cc_r == 0;
cc_v = 0;
cc_c = %r1 < %r2;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

TST: Test the value in r1

Format

TST %r1

31 24	23 16	15 8	7 0
0x0F	r1	r2	rd

Description

The `tst` instruction checks the value in **`r1`** to see if it is zero (0). Only the Z bit (cc_z) is set by this instruction, all other comparison result registers, listed in Table6.2.3 are cleared.

Pseudocode

```
cc_n = cc_v = cc_c = 0;
cc_z = %r1 == 0;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BA: branch absolute

Format

BA label

31 … 24	23 … 0
0x11	label

Description

This instruction branches to the label indicated by setting the Program Counter (`pc`) to the instruction indicated at the `label`.

Pseudocode

```
%pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BE: branch equal

Format

BE label

31 — 24	23 — 0
0x12	label

Description

The `be` instruction sets the PC to a new label if, and only if the result of the last `cmp` or `tst` set the zero bit (cc_z) to a value other than 0.

Pseudocode

```
if (cc_z)
%pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BNE: branch not equal

Format

BNE label

31 24	23 0
0x13	label

Description

The `be` instruction sets the PC to a new label if, and only if the result of the last `cmp` restulted in the zero bit (cc_z) being cleared, or set to 0.

Pseudocode

```
if (cc_z == 0)
%pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BG: branch greater than

Format

BG label

31 — 24	23 — 0
0x14	label

Description

The `bg` instruction sets the PC to a new label if, and only if the result of the last `cmp` instrucion indicated that

Pseudocode

```
if (cc_z)
%pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BGU: branch greater than, unsigned

Format

BGU label

31 … 24	23 … 0
0x15	label

Description

The `bgu` instruction sets the **`pc`** to the new label if, and only if, the result of the previous comparison shows that **`r1`** was greater than **`r2`**.

Pseudocode

```
if ((cc_c | cc_z) == 0)
pc = label;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BGE: branch greater than or equal to

Format

BGE label

31 — 24	23 — 0
0x16	label

Description

The `bge` instruction jumps to the supplied label if and only if the result of the previous comparison indicates that the value in register **`r1`** was greater than or equal to the value in **`r2`**.

Pseudocode

```
if ((cc_n ^ cc_v) == 0)
pc = label;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BGEU: branch greater than or equal to, unsigned

Format

BGEU label

31 24	23 0
0x17	label

Description

The `bgeu` instruction jumps to the supplied label if and only if the result of the previous comparison indicates that the value in register **`r1`** was greater than or equal to the value in **`r2`**.

Pseudocode

```
if (cc_c == 0)
pc = label;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BL: branch less than

Format

BL label

31 — 24	23 — 0
0x18	label

Description

The `bl` instruction jumps to the specified label if and only if the result of the previous comparison instruction indicated that the value in **`r1`** was strictly less than the value in **`r2`**.

Pseudocode

```
if (cc_n ^ cc_v)
pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BLU: branch less than, unsigned

Format

BL label

31–24	23–0
0x19	label

Description

The `blu` instruction jumps to the specified label if and only if the result of the previous comparison instruction indicated that the value in **`r1`** was strictly less than the value in **`r2`**.

Pseudocode

```
if (cc_c)
pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BLE: branch less than or equal

Format

BL label

31 — 24	23 — 0
0x1A	label

Description

The `ble` instruction jumps to the specified label if and only if the result of the previous comparison instruction indicated that the value in **`r1`** was less than, or equal to, the value in **`r2`**.

Pseudocode

```
if (cc_z | (cc_n ^ cc_v))
pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

BLEU: branch less than or equal, unsigned

Format

BLEU label

31 24	23 0
0x1B	label

Description

The `ble` instruction jumps to the specified label if and only if the result of the previous comparison instruction indicated that the value in **`r1`** was less than, or equal to, the value in **`r2`**.

Pseudocode

```
if (cc_c | cc_z)
pc = label
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDSB: load an 8 bit value

Format

LDSB %rd, %r1

31–24	23–16	15–8	7–0
0x1C	r1	r2	rd

Description

The `ldsb` instruction loads the value pointed to by **`r1`** into **`rd`**, the results register. This instruction is a **signed** instruction and will perform sign extension on the resulting register when applicable.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDSH: load a 16 bit value

Format

LDSB %rd, %r1

31 – 24	23 – 16	15 – 8	7 – 0
0x1D	r1	r2	rd

Description

The `ldsh` instruction loads a 16-bit value pointed to by **`r1`** into **`rd`**, the results register. This instruction is a **signed** instruction and will perform sign extenstion on the resulting register when applicable.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDSW: load a 32 bit value

Format

LDSB %rd, %r1

31 … 24	23 … 16	15 … 8	7 … 0
0x1E	r1	r2	rd

Description

The `ldsw` instruction loads a 32-bit value pointed to by **`r1`** into **`rd`**, the results register. This instruction is a **signed** instruction and will perform sign extension on the resulting register when applicable.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDUB: load an unsigned 8 bit value

Format

LDUB %rd, %r1

31 — 24	23 — 16	15 — 8	7 — 0
0x1F	r1	r2	rd

Description

The `ldub` instruction loads the value pointed to by **`r1`** into **`rd`**, the results register. This is an **unsigned** instruction and will not perform sign extension in any case.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDUH: load an unsigned 16 bit value

Format

LDSB %rd, %r1

31 — 24	23 — 16	15 — 8	7 — 0
0x20	r1	r2	rd

Description

The `lduh` instruction loads a 16-bit value pointed to by **`r1`** into **`rd`**, the results register. This is an **unsigned** instruction and will not perform sign extension in any case.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDUW: load an unsigned 32 bit value

Format

LDSB %rd, %r1

31 24	23 16	15 8	7 0
0x21	r1	r2	rd

Description

The `lduw` instruction loads a 32-bit value pointed to by **`r1`** into **`rd`**, the results register. This is an **unsigned** instruction and will not perform sign extension in any case.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDX: load 64 bit value

Format

LDX %rd, %r1

31–24	23–16	15–8	7–0
0x22	r1	r2	rd

Description

The `ldx` instruction loads a 64 bit value pointed to by **`r1`** into **`rd`**. Much like conventional RISC architectures, it does not perform sign extension, as this is considered to be the widest type.

Pseudocode

```
%rd = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RET: return

Format

RET %rd

31	8 7 0
0x23	rd

Description

The `ret` instruction returns the value in **`rd`**. This instruction also sets the `%pc` register to the length of the DIFO text section.

Pseudocode

```
%pc = textlen
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

NOP: no operation

Format

NOP

31	0
0x24	0

Description

The `nop` does nothing and has no side effects on the DTrace virtual machine.

Pseudocode

```
nop
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SCMP: compare two strings

Format

SCMP %r1, %r2

31 … 24	23 … 16	15 … 8	7 … 0
0x27	r1	r2	rd

Description

The `scmp` intruction compares the strings pointed to by **`r1`** and **`r2`** and sets the comparison bits for the DIF interpreter based on the result. The length of the the strings is derived by DTrace itself and the comparison is bounded by the `DTRACEOPT_STRSIZE` option set for the system.

Pseudocode

```
cc_r = strncmp(r1, r2, size);

cc_n = cc_r < 0;
cc_z = cc_r == 0;
cc_v = cc_c = 0;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDGA: load a DTrace built-in variable

Format

LDGA %rd, var, %r2

31 24	23 16	15 8	7 0
op	var	r2	rd

Description

The `ldga` instruction looks up the value of a DTrace built-in variable based on the value in **`var`** with an optional array index in the register `%r2`.

Unlike the `ldgs`, the variable identifier is 8 bits long, and the other 8 bits are used to identify the register which contains the index of the array.

Pseudocode

```
index = %r2
%rd = var[index]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDGS: Load a user defined variable

Format

LDGS %rd, %r1, %r2

31 24	23 8	7 0
0x29	var	rd

Description

The `ldgs` instruction has two modes of operation and is intended to be used only for scalar values. The first mode of operation is when the value provided in **`var`** is less than DIF_VAR_OTHER_UBASE. This will cause DTrace to look up a pre-defined scalar variable such as curthread, while the second mode of operation will result in looking up a user defined variable in a DIF program. The result of this instruction will be put into the register **`rd`**.

Unlike the `ldga` instruction, the **`var`** field is 16 bits long, as opposed to 8 bits due to the fact that the variable that is being loaded is a scalar and does not require indexing operations.

Pseudocode

```
%rd = var
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STGS: store a value into a variable

Format

STGS %rd, %r1, %r2

31 … 24	23 … 8	7 … 0
0x2A	var	rd

Description

Similar to `ldgs`, the instruction `stgs` operates exclusively on scalar variables and can not contain indices. However, the instruction may allow loading of data by reference using the DIF_TF_BYREF flag, which allows loading of data bounded by the limits found in the dtrace_vcanload() function. Unlike `ldgs`, `stgs` can not store to pre-defined variables in DTrace, and instead allows access only to user defined variables. The variable is accessed by the **`var`** field and is required to be large or equal to DIF_VAR_OTHER_UBASE. The result of this operation is stored in the **`rd`** register.

Pseudocode

```
assert(var >= DIF_VAR_OTHER_UBASE)
var -= DIF_VAR_OTHER_UBASE
if (flags & DIF_TF_BYREF)
    var = copyin(%rd)
else
    var = %rd
```

Constraints

Failure modes

This instruction will fail if the supplied value in the **`var`** field is less than DIF_VAR_OTHER_UBASE.

LDTA: Load thread local array UNIMPLEMENTED

Format

LDTA %rd, var, %r2

31 24	23 16	15 8	7 0
0x2B	var	r2	rd

Description

The `ldta` instruction is unimplemented and reserved for future use.

Pseudocode

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDTS: load a value from a thread local variable

Format

LDTS %rd, %r1, %r2

31 24	23 8	7 0
0x2C	variable	rd

Description

The `ldts` instruction loads data from a thread local variable into the **`rd`** register by reference or by value. The `DIF_TF_BYREF` flag is used to determine the appropriate lookup.

Pseudocode

```
%rd = var
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STTS: Store a value into thread local storage

Format

STTS %rd, %r1, %r2

31 — 24	23 — 8	7 — 0
0x2D	variable	rd

Description

The `stts` instruction takes the value stored in **`rd`** and stores it directly, or by reference into a thread local variable. The `DIF_TF_BYREF` flag is used to determine the appropriate lookup.

Pseudocode

```
var = %rd
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SRA: Shift Right Arithmetic

Format

SRA %rd, %r1, %r2

31 … 24	23 … 16	15 … 8	7 … 0
0x2E	r1	r2	rd

Description

The `sra` instruction shifts the value in **`r1`** right by the number of bits indicated in **`r2`**, placing the results in register **`rd`**. This instruction only operates on **signed** integers.

Pseudocode

```
%rd = %r1 >> %r2
```

Constraints

`r1`, **`r2`**, and **`rd`** must be less than **`NREGS`**.

`rd` cannot be `%r0`.

Failure modes

This instruction has no run-time failure modes beyond its constraints.

PUSHTR: push a reference onto the stack

Format

PUSHTR type, %r2, %rs

31 24	23 16	15 8	7 0
0x30	type	r2	rs

Description

The `pushtr` instruction pushes a reference, contained in the **`rs`** register onto the stack. The length is stored for a string along with the value. For a numeric value the size of that value is stored.

Pseudocode

```
value = %rs
if type is string:
    size = strlen(value)
else:
    size = %r2

stack[++index].size = size
stack[index].value = value
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

PUSHTV: push a value onto the stack

Format

PUSHTV %rs

31 24	23 16	15 8	7 0
0x31	r0	r0	rs

Description

The `pushtv` instruction takes the value contained in **`rs`** register and pushes it onto the stack. Unlike the `PUSHTR` instruction, the size of the value is *not* stored along with the value.

Pseudocode

```
stack[++index].value = %rs
stack[index].size = 0;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

POPTS: pop a value from the stack

Format

POPTS

31	8 7 0
0x32	0

Description

The `popts` pops the stack, moving the stack's index to next position down from the top, without returning any value.

Pseudocode

```
stack[index--]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

FLUSHTS: flush the stack

Format

FLUSHTS

31	8 7 0
0x33	0

Description

The `flushts` instruction flushes the stack, by resetting the stack pointer to 0.

Pseudocode

```
%sp = 0;
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ALLOCS: allocate a string

Format

ALLOCS %rd, %r1

31 — 24	23 — 16	15 — 8	7 — 0
0x3A	r1	r2	rd

Description

The `allocs` instruction allocates a string in the DIF scratch space, based on the size in **`r1`** and returns the pointer to that string in register **`rd`**. A failed allocation returns a 0.

Pseudocode

```
ptr = scratch_space;
scratch_space += size;
%rd = ptr
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

COPYS: copy a string

Format

COPYS %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x3B	r1	r2	rd

Description

The `copys` instruction copies bytes from the string pointed to by **`r1`** and returns them in **`rd`** bounded by a size placed into **`r2`**.

Pseudocode

```
%rd = copy(r1, r2)
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STB: store a byte into memory

Format

STB %rd, %r1

31 – 24	23 – 16	15 – 8	7 – 0
0x3C	r1	r2	rd

Description

The `stb` instruction takes a byte from **`r1`** and stores it into the memory location pointed to by **`rd`**.

Pseudocode

```
mem[%rd] = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STH: store a 16 bit value into memory

Format

STH %rd, %r1

31 24	23 16	15 8	7 0
0x3D	r1	r2	rd

Description

The `sth` instruction takes a 16 bit value from **`r1`** and stores it into the memory location pointed to by **`rd`**.

Pseudocode

```
mem[%rd] = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STW: store a 32 bit value into memory

Format

STW %rd, %r1

31 — 24	23 — 16	15 — 8	7 — 0
0x3E	r1	r2	rd

Description

The `stw` instruction takes a 32 bit value from **`r1`** and stores it into the memory location pointed to by **`rd`**.

Pseudocode

```
mem[%rd] = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STX: store a 64 bit value into memory

Format

STX %rd, %r1

31 24	23 16	15 8	7 0
0x3F	r1	r2	rd

Description

The `stx` instruction takes a 64 bit value from **`r1`** and stores it into the memory location pointed to by **`rd`**.

Pseudocode

```
mem[%rd] = %r1
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDSB: load signed 8 bit quantity from user space

Format

ULDSB %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x40	r1	r2	rd

Description

The `uldsb` instruction loads a signed 8 bit quantity from memory in a user space process into the **`rd`** register, indexed by **`r1`**. This instruction is a **signed** instruction and will perform sign extension on the resulting register when applicable.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDSH: load a signed 16 bit quantity from user space

Format

ULDSH %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x41	r1	r2	rd

Description

The `uldsh` instruction loads a signed, 16 bit, quantity from memory in a user space process into the **`rd`** register, indexed by **`r1`**. This instruction is a **signed** instruction and will perform sign extension on the resulting register when applicable.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDSW: load a signed 32 bit quantity from user space

Format

ULDSW %rd, %r1, %r2

31 — 24	23 — 16	15 — 8	7 — 0
0x42	r1	r2	rd

Description

The `uldsw` instruction loads a signed 32 bit quantity from memory in a user space process into the **`rd`** register, indexed by **`r1`**. This instruction is a **signed** instruction and will perform sign extension on the resulting register when applicable.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDUB: load unsigned 8 bit quantity from user space

Format

ULDUB %rd, %r1, %r2

31 — 24	23 — 16	15 — 8	7 — 0
0x43	r1	r2	rd

Description

The `uldub` instruction loads a unsigned 8 bit quantity from memory in a user space process into the **`rd`** register indexed by **`r1`**. This is an **unsigned** instruction and will not perform sign extension in any case.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDUH: load an unsigned 16 bit quantity from user space

Format

ULDUH %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x44	r1	r2	rd

Description

The `ulduh` instruction loads an unsigned, 16 bit, quantity from memory in a user space process into the **`rd`** register, indexed by **`r1`**. This is an **unsigned** instruction and will not perform sign extension in any case.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDUW: load an unsigned 32 bit quantity from user space

Format

ULDUW %rd, %r1, %r2

31 – 24	23 – 16	15 – 8	7 – 0
0x45	r1	r2	rd

Description

The `ulduw` instruction loads an unsigned 32 bit quantity from memory in a user space process into the **`rd`** register, indexed by **`r1`**. This is an **unsigned** instruction and will not perform sign extension in any case.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

ULDX: load a 64 bit value from user program memory

Format

ULDX %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x46	r1	r2	rd

Description

The `uldx` instruction loads a 64 bit value from a user space program's memory into the **`rd`** register, indexed by **`r1`**. Much like conventional RISC architectures, it does not perform sign extension, as this is considered the widest type.

Pseudocode

```
%rd = umem[r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDSB: restricted load of a signed 8 bit quantity

Format

RLDSB %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x47	r1	r2	rd

Description

The `rldsb` instruction performs a privilege check on the memory it is about to read from before loading a signed, 8 bit, quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDSH: restricted load of a signed 16 bit quantity

Format

RLDSH %rd, %r1, %r2

31–24	23–16	15–8	7–0
0x48	r1	r2	rd

Description

The `rldsh` instruction performs a privilege check on the memory it is about to read from before loading a signed, 16 bit, quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDSW: restricted load of a signed 32 bit quantity

Format

RLDSW %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x49	r1	r2	rd

Description

The `rldsw` instruction performs a privilege check on the memory it is about to read from before loading a signed, 32 bit, quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDUB: restricted load of an unsigned 8 bit quantity

Format

RLDUB %rd, %r1, %r2

31 – 24	23 – 16	15 – 8	7 – 0
0x4A	r1	r2	rd

Description

The `rldub` instruction performs a privilege check on the memory it is about to read from before loading an unsigned, 8 bit, quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDUH: restricted load of an unsigned 16 bit quantity

Format

RLDUH %rd, %r1, %r2

31 24	23 16	15 8	7 0
0x4B	r1	r2	rd

Description

The `rlduh` instruction performs a privilege check on the memory it is about to read from before loading an unsigned, 16 bit, quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDUW: restricted load of an unsigned 32 bit quantity

Format

RLDUW %rd, %r1, %r2

31–24	23–16	15–8	7–0
0x4C	r1	r2	rd

Description

The `rlduw` instruction performs a privilege check on the memory it is about to read from before loading an unsigned, 32 bit, quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

RLDX: restricted load of a 64 bit quantity

Format

RLDX %rd, %r1, %r2

31 … 24	23 … 16	15 … 8	7 … 0
0x4D	r1	r2	rd

Description

The `rldx` instruction performs a privilege check on the memory it is about to read from before loading a 64 bit quantity into **`rd`**, indexed by **`r1`**.

Pseudocode

```
%rd = mem[%r1]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SETX: retrieve an integer from the integer table

Format

SETX %rd, intindex

31	16	15 8 7 0
0x25	rd	index

Description

The `setx` instruction looks up an integer value stored in the DIF integer table and places it into **`rd`**. This instruction performs no bounds checking.

Pseudocode

```
%rd = inttab[index]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

SETS: retrieve string from the string table

Format

SETS %rd, strindex

31	16	15 8	7 0
0x26	rd	index	

Description

The `sets` instruction looks up a string stored in the DIF string table and places a pointer to the value into **`rd`**. This instruction performs no bounds checking.

Pseudocode

```
%rd = strtab + index
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

CALL: subroutine call

Format

CALL %rd, %r1, %r2

31 24	23 8	7 0
0x2F	subroutine	rd

Description

The `call` instruction executes a known DTrace subroutine, such as copyinstr(), copyout() etc. and returns any value into **`rd`**. Valid subroutines are documented in 6.2.5.

Pseudocode

```
%rd = subr()
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDGAA: load a value from a hash map

Format

LDGAA key, %rd

31 24	23 8	7 0
0x34	key	rd

Description

The `ldgaa` instruction loads a value into the **`rd`** register based on a key. The key is used to lookup the value in a hash map data structure.

Pseudocode

```
%rd = map[key]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDTAA: load a value from a thread private hash map

Format

LDTAA var, %rd

31 24	23 8	7 0
0x35	key	rd

Description

The `ldtaa` instruction loads a value into the **`rd`** register based on a key. The key is used to lookup the value in a thread private, hash map, data structure.

Pseudocode

```
%rd = map[key]
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STGAA: store a value into a hash by key

Format

STGAA key, %rd

31–24	23–8	7–0
0x46	key	rd

Description

The `stgaa` instruction stores a value, contained in the **`rd`** register into a hash map based on a key.

Pseudocode

```
map[key] = %rd
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STTAA: store a value into a thread private, hash by key

Format

STTAA key, %rd

31 24	23 8	7 0
0x47	key	rd

Description

The `sttaa` instruction stores a value, contained in the **`rd`** register into a thread private, hash map based on a key.

Pseudocode

```
map[key] = %rd
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

LDLS: load local variable

Format

LDLS variable, %rd

31 — 24	23 — 8	7 — 0
0x48	variable	rd

Description

The `ldls` instruction loads a local variable into the **`rd`** register.

Pseudocode

```
%rd = var
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

STLS: store a value in a local variable

Format

STLS variable, %rd

31 24	23 8	7 0
0x49	variable	rd

Description

The `stls` instruction takes a value from the **`rd`** register and stores it in a variable.

Pseudocode

```
var = %rd
```

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

XLATE:

Format

XLATE %rd, %r1, %r2

31 — 24	23 — 16	15 — 8	7 — 0
0x4E	r1	r2	rd

Description

The `xlate` instruction extracts translated data indicated at current the current translation index and returns the data in **`rd`**.

Pseudocode

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

XLARG: translation argument

Format

XLARG %rd, %r1, %r2

31 — 24	23 — 16	15 — 8	7 — 0
0x4F	r1	r2	rd

Description

The `xlarg` instruction translates a single argument from a structure and returns the tranlsated value in **`rd`**.

Pseudocode

Constraints

Failure modes

This instruction has no run-time failure modes beyond its constraints.

Chapter 8

Variable Records

Chapter 9

Subroutines

9.1 Subroutine calling mechanism

9.2 Subroutine list

9.3 Subroutine reference

Name	Number	Description
rand	0	Get random
mutex_owned	1	Query whether current thread is mutex owner
mutex_owner	2	Retrieve mutex owner
mutex_type_adaptive	3	Query if mutex is adaptive
mutex_type_spin	4	Query if mutex is a spinlock
rw_read_held	5	Query whether rwlock is held for read
rw_write_held	6	Query whether current thread holds rwlock for write
rw_iswriter	7	Query whether rwlock is held for write
copyin	8	Copy in data from userspace
copyinstr	9	Copy in string from userspace
copyoutmbuf	9	Copy data from an mbuf chain
speculation	10	
progenyof	11	
strlen	12	
copyout	13	
copyoutstr	14	
alloca	15	
bcopy	16	
copyinto	17	
msgdsize	18	
msgsize	19	
getmajor	20	
getminor	21	
ddi_pathname	22	
strjoin	23	
lltostr	24	
basename	25	
dirname	26	
cleanpath	27	
strchr	28	

Table 9.1: DTrace Subroutines (Part 1)

Name	Number	Description
`strrchr`	29	
`strstr`	30	
`strtok`	31	
`substr`	32	
`index`	33	
`rindex`	34	
`htons`	35	
`htonl`	36	
`htonll`	37	
`ntohs`	38	
`ntohl`	39	
`ntohll`	40	
`inet_ntop`	41	
`inet_ntoa`	42	
`inet_ntoa6`	43	
`toupper`	44	
`tolower`	45	
`memref`	46	
`sx_shared_held`	48	
`sx_exclusive_held`	49	
`sx_isexclusive`	50	
`memstr`	51	
`getf`	52	
`json`	53	
`strtoll`	54	
`random`	55	
`uuidstr`	56	

Table 9.2: DTrace Subroutines (Part 2)

rand(): Get Random

Calling convention

`rd` Target for 64 bits of random(ish) data

Description

This subroutine returns 64 bits of random(ish) data, placing the result in **`rd`**. On supporting systems, stronger randomness can be obtained uing the `random` subroutine.

Pseudocode

```
regs[rd] = (getthrtime() * 2416 + 374441) % 1771875
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

mutex_owned: Is this mutex owned by a thread

Calling convention

rd Boolean value indicating mutex ownership.

Description

The `mutex_owned` subroutine takes a mutex as its argument and returns a boolean value indicating whether the mutex is currently owned by a thread.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

mutex_owner: Report which thread owns a mutex

Calling convention

`retval` The kernel thread which owns the mutex

`arguments`

Description

The `mutex_owner` subroutine returns the kernel thread structure which owns the mutex passed at the only argument.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

mutex_type_adaptive: Is the mutex adaptive

Calling convention

`retval` Boolean indication of whether or not the mutex is adaptive.

Description

The `mutex_type_adaptive` subroutine takes a mutex as its only arugment and returns a boolean value indicating whether or not the mutex is adaptive.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

mutex_type_spin: Spin mutex detection

Calling convention

`rd` Boolean value indicating whether or not the mutex passed as this subroutine's only argument is a spin mutex.

Description

The `mutex_type_spin` subroutine takes a mutex as its only arugment and returns a boolean value indicating wether or not the mutex is a spin mutex.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

rw_read_held: Is this read/write mutex currently held by a reader

Calling convention

`rd` Boolean value indicating if this read/write mutex is currently held.

Description

The `rw_read_held` subroutine takes a read/write mutex as its only argument and returns a boolean value indicating if the mutex is currently held by a reader.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

rw_write_held: Is this read/write mutex held by a writer

Calling convention

`rd` Boolean value indicating whether or not a read/write mutex is held by a writer.

Description

The `rw_write_held` subroutine takes a read/write mutex as its only argument and returns a boolean value indicating whether or not the mutex is held by a writer.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

rw_iswriter: Does the current thread hold a r/w mutex as a writer

Calling convention

`rd` Boolean value indicating if the current thread holds a read/write mutex as a writer.

Description

The `rw_iswriter` function takes a read/write mutex as its only arugment and returns a boolean value indicating if the current rhead holds the mutex as a writer.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

copyin: Copy data from user space to kernel space

Calling convention

`rd` Pointer to kernel data.

Description

The `copyin` returns a pointer to a buffer which contains kernel data copied from the area pointed to by its first argument, up to the limit denoted by its second argument.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

copyinstr: Copy kernel data as a string

Calling convention

`rd` Pointer to the returned string.

Description

The `copyinstr` subroutine returns a pointer to string of kernel data which is located at the first argument and bounded by the second argument.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

speculation: Activate an inactive speculation

Calling convention

`rd` Either an active speculation or 0.

Description

The `speculation` subroutine transitions an inactive speculation to the active state, and returns it to the caller, or returns 0 if there are no inactive speculations available.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

progenyof:is this process the child of a particular PID

Calling convention

`rd` Boolean value

Description

The `progenyof` subroutine returns a boolean value that indicates if the current process is a child of the PID passed in the only argument.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strlen: DTrace version of the strlen function

Subroutine prototype

```
size_t strlen(const char *string);
```

Calling convention

arg0 Pointer to the string

rd Length of the string passed as the only argument

Description

The `strlen` subroutine is DTrace's version of the well known C library function. It returns the length, in bytes, of the string pointed to by the pointer passed in as its first argument. The string must be NULL terminated.

Pseudocode

```
string = stack[0].value
%rd = strlen(string)
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

copyout: FILL ME IN

Calling convention

`rd` What goes into the destination regiester

Description

This subroutine

Pseudocode

```
regs[rd] = (getthrtime() * 2416 + 374441) % 1771875
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

copyoutstr: copy data from kernel to user space, as a string

Calling convention

`rd` VOID

Description

The `copyoutstr` subroutine copies data from kernel space to user space as a string value, bounded by the routine's third argument.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

copyoutmbuf: copy data from an mbuf chain

Calling convention

`arg0` pointer to mbuf

`arg1` amount of data to copy

`rd` pointer to copied data

Description

The `copyoutmbuf` subroutine copies data from an mbuf chain out a destination pointer bounded by a size given in the second argument. If the second argument exceeds the size of the data in the mbuf chain then it is reduced to the correct length.

Pseudocode

Constraints

The `copyoutmbuf` subroutine is only supported on FreeBSD.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

alloca: allocate temporary space

Calling convention

`rd` Pointer to allocated data or NULL.

Description

The `alloca` subroutine allocates scratch space in the DTrace state machine structure. Although this subroutine does not allocate space on the process stack, it does act similarly to the alloca macro, in that the space disappears without an explicit call to a `free` routine, once the DTrace machine state structure is deallocated.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

bcopy: copy bytes from source to destination bounded by a size

Subroutine prototype

```
void bcopy(const void *source, void *destination, size_t length);
```

Calling convention

arg0 Pointer to the source memory

arg1 Pointer to the destination scratch memory

arg2 Amount of bytes to copy

rd VOID

Description

The `bcopy` subroutine copies bytes from a source pointer to a destination pointer, within the DTrace machine state scratch region, up to the size supplied in the third argument.

Pseudocode

```
source = stack[0].value
destination = stack[1].value
length = stack[2].value

if destination not in scratch:
    return

if not can_load(source):
    %rd = 0
    return

for i = 0 ... length:
    destination[i] = source[i]
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

copyinto: copy data from a source to a destination

Calling convention

rd VOID

Description

The `copyinto` subroutine copies data from a source pointer into a destination pointer bounded by a size given in the second argument.

Pseudocode

```
regs[rd] = (getthrtime() * 2416 + 374441) % 1771875
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

msgdsize:

Calling convention

`rd` What goes into the destination regiester

Description

This subroutine

Pseudocode

Constraints

The `msgdsize` subroutine is only available on the Illumos operating system.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

msgsize: FILL ME IN

Calling convention

`rd` What goes into the destination regiester

Description

Pseudocode

Constraints

The `msgsize` subroutine is only available on the Illumos operating system.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

getmajor: return major device number

Calling convention

`rd` Major device number

Description

The `getmajor` subroutine returns the major device number from a device structure supplied as the first argument.

Pseudocode

Constraints

The `getmajor` subroutine is only available on Illumos and systems derivice from OpenSolaris.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

getminor: Get the minor device number from a device structure

Calling convention

`rd` Minor device number

Description

The `getminor` subroutine returns the minor number from a device structure.

Pseudocode

Constraints

The `getminor` subroutine is only available on Illumos and systems derivice from OpenSolaris.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

ddi_pathname: look up device driver by name

Calling convention

arg0 Pointer to a device node.

arg1 Device minor number.

rd Path within the /devices tree.

Description

The `ddi_pathname` subroutine returns a string describing the device driver that implements a device in the system.

Pseudocode

Constraints

The `ddi_pathname` subroutine is only available on Illumos and systems derivice from OpenSolaris.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strjoin: joing two strings and return the result

Subroutine prototype

```
char * strjoin(const char *first, const char *second);
```

Calling convention

`arg0` Pointer to the first string

`arg1` Pointer to the second string

`rd` Pointer to the combined string

Description

The `strjoin` subroutine concatenates the two strings passed to it as arguments and returns the combined string to the caller.

Pseudocode

```
first = stack[0].value
second = stack[1].value
combined = scratch_space

if (not can_load(first)) or (not can_load(second)):
    %rd = 0
    return

if no room in scratch:
    %rd = 0
    return

for i = 0 ... len(first):
    combined[i] = first[i]

for j = 0 ... len(second):
    combined[i + j] = second[j]

scratch_space += len(combined)
%rd = combined
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

lltostr: convert a long long (64 bit) value to a string

Calling convention

`rd` string representation of passed value

Description

The `lltostr` subroutine takes a 64 bit value as its only argument and returns that value as a human readable string.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

basename: return the file name portion of a pathname

Calling convention

`rd` A pointer to a scratch space string containing the filename.

Description

The `basename` subroutine takes a single string argument, containing a path, and returns a pointer to the file name portion of the supplied string. The space for the resulting string is contained in the DTrace machine state structure, `mstate` which is automatically de-allocated.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

dirname: return the directory component of a pathname

Calling convention

`rd` A string pointing to the directory component of a pathname.

Description

The `dirname` subroutine returns a string containing the directory component of a pathname, without the terminating filename.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

cleanpath: return the cleaned up pathname

Calling convention

`rd` A pointer to the scratch space string containing the cleaned up pathname.

Description

The `cleanpath` subroutine takes a single string argument, containing a path, and returns a pointer to a string containing the cleaned up pathname.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strchr: locate a character in a string

Calling convention

rd pointer to the character or NULL if not found

Description

The `strchr` subroutine searches a string, supplied as the first argument, for the first instance of the character passed as the second and returns a pointer to the location of the character in the string. If the charaacter is not present in the string then NULL is returned.

Pseudocode

```
addr = stack[0].value
target = stack[1].value
%rd = strchr(addr, target)
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strrchr: reverse search a string

Calling convention

rd pointer to the character or NULL if not found

Description

The `strrchr` subroutine searches a string, supplied as the first argument, for the last instance of the character passed as the second and returns a pointer to the location of the character in the string. If the charaacter is not present in the string then NULL is returned.

Pseudocode

```
addr = stack[0].value
target = stack[1].value
%rd = strrchr(addr, target)
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strstr: locate a string within a string

Subroutine prototype

```
char * strstr(const char *big, const char *little);
```

Calling convention

arg0 Pointer to the string to be searched through

arg1 Pointer to the string to search for

rd Pointer to the string located or NULL if not found

Description

The `strstr` subroutine search a string, passed as its first argument, for a sub-string, passed as the second argument. If the sub-string is found a pointer to it is returned to the caller, otherwise NULL is returned.

Pseudocode

```
big = stack[0].value
little = stack[1].value
%rd = strstr(big, little)
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strtok: string tokenizing subroutine

Calling convention

rd pointer to the next token or NULL

Description

The `strtok` subroutine returns a sequential set of tokens from a string passed as its first argument, based on a separator passed as its second. Once the string has been exhausted NULL is returned. In order to find subsequent tokens NULL is passed as the first argument. See this operating system's `strtok` manual page (strtok(3)) for an example.

Pseudocode

```
string = stack[0].value
separator = stack[1].value
%rd = strtok(string, separator)
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

substr: return a sub string of a string

Calling convention

`rd` a string representing the substring

Description

The `substr` routine returns a sub-string of a string, passed as the first argument, starting from a byte index passed as the second argument. An optional third argument can be used to bound the resulting string. If the optional bounding argument is not supplied then the sub-string includes all bytes up to and including the terminating NUL character.

Pseudocode

```
string = stack[0].value
index = stack[1].value
length = stack[1].value
%rd = substr(string, index, length)
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

index: return the byte position of a character in a string

Calling convention

`rd` Position of character or -1

Description

The `index` subroutine searches from the beginning of a string pointed to by its first argument, for a character supplied as the second argument. The search proceeds until the character is found, or an optional limit, supplied as the third argument is reached. If the character is not found then -1 is returned to the caller.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

rindex: locate the last matching character in a a string

Calling convention

`rd` The position of the character or `-1` if the character is not found.

Description

The `rindex` subroutine searches from the end of a string pointed to by its first argument, for the first instance character supplied as its second argument. The search proceeds until the character is found, or an optional limit, supplied as the third argument is reached. If the character is not found then `-1` is returned to the caller.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

htons: convert a short (16 bit) value from host to network byte order

Calling convention

`rd` A 16 bit value in network byte order

Description

The `htons` subroutine takes a 16 bit value as its only argument and returns that value in network byte order.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

htonl: convert a long (32 bit) value from host to network byte order

Calling convention

`rd` Long value in network byte order

Description

The `htonl` subroutine takes a long value as its only argument and returns the same long value in network byte order, suitable for use in network protocols.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

hotnll: convert a long long (64 bit) value from host to network byte order

Calling convention

`rd` A 64 bit value in network byte order

Description

The `htonll` routine takes a 64 bit value as its only argument and returns that value in network byte order.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

ntohs: convert a short (16 bit) value from network to host byte order

Calling convention

`rd` short (16 bit) value in host byte order

Description

The `ntohs` subroutine takes a short (16 bit) value as its only argument and returns the same value in host byte order.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

ntohl: convert long (32 bit) value from network to host byte order

Calling convention

`rd` value in host byte order

Description

The `ntohl` routine takes a 32 bit value as its only argument and returns that value in host byte order.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

ntohll: convert a long long (64 bit) value from network to host byte order

Calling convention

`rd` long long (64 bit) value in host byte order

Description

The `ntohll` subroutine takes a long long (64 bit) value as its only argument and returns that value in host byte order.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

inet_ntop: convert an arbitrary Internet address to a string

Calling convention

`rd` Internet address as a string

Description

The `inet_ntop` subroutine takes either a 128 bit, IPv6, address or a 32 bit, IPv4 address, and converts it to a string suitable for humans. The type of address supplied is indeicated by the second argument, which must either be `AF_INET` or `AF_INET6`.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

inet_ntoa: convert a 32 bit IPv4 address to a string

Calling convention

`rd` IPv4 address as a string

Description

The `inet_ntoa` subroutine takes a 32 bit, IPv4, address and converts it to a string suitable for humans.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

inet_ntoa6: convert a 128 bit IPv6 address to a string

Calling convention

`rd` IPv6 address as a string

Description

The `inet_ntoa6` subroutine takes a 128 bit, IPv6, address and converts it to a string suitable for humans.

Pseudocode

```
regs[rd] = (getthrtime() * 2416 + 374441) % 1771875
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

toupper: convert a string to upper case

Subroutine prototype

```
char * toupper(const char *string);
```

Calling convention

`arg0` Pointer to the string

`rd` A string with only upper case letters

Description

The `toupper` subroutine converts the characters of the string supplied as its only argument into upper case and returns the resulting string.

Pseudocode

```
string = stack[0].value
destination = scratch_space

for i = 0 ... len(string):
    c = string[i]
    if c is lowercase:
        c = uppercase(c)
    destination[i] = c

scratch_space += len(string)
%rd = destination
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

tolower: convert a string to all lower case characters

Subroutine prototype

```
char * tolower(const char *string);
```

Calling convention

`arg0` Pointer to the string

`rd` An all lower case string

Description

The `tolower` subroutine returns a string converts the characters of the string supplied as its only argument into lower case and returns the resulting string.

Pseudocode

```
string = stack[0].value
destination = scratch_space

for i = 0 ... len(string):
    c = string[i]
    if c is uppercase:
        c = lowercase(c)
    destination[i] = c

scratch_space += len(string)
%rd = destination
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

memref: return scratch memory

Subroutine prototype

```
uintptr_t * memref(uintptr_t ptr, size_t length);
```

Calling convention

arg0 Pointer to memory

arg1 Length of scratch memory to use

rd Pointer to a fixed size of scratch memory

Description

The `memref` subroutine allocates memory from scratch space and returns that memory to the caller.

Pseudocode

```
size = sizeof(uintptr_t) * 2
memref = scratch_space
memref[0] = stack[0].value
memref[1] = stack[1].value
scratch_space += size
%rd = memref
```

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

sx_shared_held: Is this shared mutex currently held by a reader

Calling convention

`rd` Boolean value indicating if this read/write mutex is currently held.

Description

The `sx_shared_held` subroutine takes an sx shared mutex as its only argument and returns a boolean value indicating if the mutex is currently held by a reader.

Pseudocode

Constraints

The `sx_shared_held` subroutine is only available on Illumos and systems derivice from OpenSolaris.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

sx_exclusive_held: Is this sx mutex held exclusively

Calling convention

`rd` Boolean value indicating whether or not a the mutex is held exclusively..

Description

The `sx_exclusive_held` subroutine takes an sx shared mutex as its only argument and returns a boolean value indicating whether or not the mutex is held exclusively.

Pseudocode

Constraints

The `sx_exclusive_held` subroutine is only available on Illumos and systems derivice from OpenSolaris.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

sx_isexclusive: Is the current thread the only one to hold a shared mutex

Calling convention

`rd` Boolean value indicating if the current thread is the only one holding a shared mutex.

Description

The `sx_isexclusie` subroutine takes a shared mutex as its only arugment and returns a boolean value indicating if the current thread is the only one holding it.

Pseudocode

Constraints

The `sx_isexclusive` subroutine is only available on Illumos and systems derivice from OpenSolaris.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

memstr: FILL ME IN

Calling convention

rd What goes into the destination regiester

Description

This subroutine

Pseudocode

```
regs[rd] = (getthrtime() * 2416 + 374441) % 1771875
```

Constraints

The `memstr` subroutine is only available on the FreeBSD operating system.

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

getf: Return a file structure based on a file descriptor

Calling convention

rd Pointer to a valid file structure.

Description

The `getf` subroutine takes a file descriptor as its argument and returns a file pointer based on the supplied file descriptor.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

json: extract a single value from a JSON string

Calling convention

`rd` A string containing the value or NULL

Description

The `json` subroutine

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

strtoll: convert a string representing a number to a long long (64 bit) value

Calling convention

`rd` a long long (64 bit) value

Description

The `strtoll` takes a number encoding in a string and converts it to a long long (64 bit) value.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

random: return a better pseudo-random number than rand()

Calling convention

`rd` A pseudo-random number

Description

The `random` subroutine returns a better pseudo-random number than the originala `rand` subroutine provided by DTrace.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

uuidstr: convert a UUID to a string

Calling convention

`rd` string representation of a UUID

Description

The `uuidstr` subroutine converts a numeric UUID into a string.

Pseudocode

Constraints

Failure modes

This subroutine has no run-time failure modes beyond its constraints.

Appendix A

Code Organization

DTrace was originally developed on `OpenSolaris` which had a unique way of organizing code. One key thing to note is that there are *two different* `dtrace.h` include files, one for the kernel and one for the user space code.

A.1 Illumos

The original source of DTrace came from OpenSolaris which has morphed into Illumos. As this was the original place that the code resided there was no reason to split things along OS or license boundaries. The main DTrace command resides in `cmd`, the supporting libraries are in `lib/libdtrace` and the kernel code is in the `uts/common`, `uts/intel`, `uts/sparc`, and related directories.

A.2 FreeBSD

Within FreeBSD the DTrace code has been split between that which came from Sun's OpenSolaris (now Illumos) and is therefore under the CDDL and the code which has been written natively on FreeBSD, and is therefore under a BSD license. There are two locations for the cddl code, one in the root of the tree, `/usr/src` and one in the kernel directory `/usr/src/sys`. Native FreeBSD scripts are located in the `/usr/share/dtrace` directory.

Because of the user space and kernel split for the cddl code the FreeBSD tree has three, separate, `dtrace.h` files:

A.3 Mac OS

Open source code from Apple is supplied in discrete pacakges. The DTrace code on Mac OS is split between the `xnu` kernel and the rest of the code which is contained in a `dtrace` code drop. The kernel includes a very small number of files that are absolutely necessary to build the kernel itself, including the driver code. All of the kernle code is collected into the `xnu/bsd/dev/dtrace/`

sys/cddl/contrib/opensolaris/uts/common/sys/dtrace.h The one you care about.

cddl/contrib/opensolaris/lib/libdtrace/common/dtrace.h Library APIs

cddl/compat/opensolaris/include/dtrace.h Compatability include

Figure A.1: The various versions of dtrace.h

directory with the Mac OS translators, the D files that know about the internals of kernel data structures, are contained in the `scripts` sub-directory. In the `OpenDTrace` repositories there Mac OS kernel code resides in `https://github.com/opendtrace/xnu` while the rest of the code resides in `https://github.com/opendtrace/macos-dtrace`. These repositories are updated as soon as Apple drops their tarballs onto `https://opensource.apple.com/tarballs/`.

Bibliography

[1] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[2] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 27 - July 2, 2004, Boston Marriott Copley Place, Boston, MA, USA*, pages 15–28, 2004.

[3] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.