Rocky Enterprise Linux 9.2 Manual Pages on command 'ltrace.conf.5'

C:\>man ltrace.conf.5

ltrace.conf(5) ltrace configuration file ltrace.conf(5)

NAME

ltrace.conf - Configuration file for ltrace(1).

DESCRIPTION

This manual page describes ltrace.conf, a file that describes prototypes of func? tions in binaries for ltrace(1) to use. Ltrace needs this information to display function call arguments.

Each line of a configuration file describes at most a single item. Lines composed entirely of white space are ignored, as are lines starting with semicolon character (comment lines). Described items can be either function prototypes, or definitions of type aliases.

PROTOTYPES

A prototype describes return type and parameter types of a single function. The syntax is as follows:

LENS NAME ([LENS{,LENS}]);

NAME is the (mangled) name of a symbol. In the elementary case, LENS is simply a type. Both lenses and types are described below. For example, a simple function prototype might look like this:

int kill(int,int);

Despite the apparent similarity with C, ltrace.conf is really its own language that's only somewhat inspired by C.

TYPES

Ltrace understands a range of primitive types. Those are interpreted according to C convention native on a given architecture. E.g. ulong is interpreted as 4-byte unsigned integer on 32-bit GNU/Linux machine, but 8-byte unsigned integer on 64-bit GNU/Linux machine.

void Denotes that a function does not return anything. Can be also used to con?

struct a generic pointer, i.e. pointer-sized number formatted in hexadecimal format.

char 8-bit quantity rendered as a character

ushort,short

Denotes unsigned or signed short integer.

uint,int

Denotes unsigned or signed integer.

ulong,long

Denotes unsigned or signed long integer.

float Denotes floating point number with single precision.

double Denotes floating point number with double precision.

Besides primitive types, the following composed types are possible:

struct([LENS{,LENS}])

Describes a structure with given types as fields, e.g. struct(int,int,float).

Alignment is computed as customary on the architecture. Custom alignment (e.g. packed structs) and bit-fields are not supported. It's also not pos? sible to differentiate between structs and non-POD C++ classes, for arches where it makes a difference.

array(LENS,EXPR)

Describes array of length EXPR, which is composed of types described by LENS, e.g. array(int, 6).

Note that in C, arrays in role of function argument decay into pointers. Ltrace currently handles this automatically, but for full formal correct? ness, any such arguments should be described as pointers to arrays.

LENS* Describes a pointer to a given type, e.g. char* or int***. Note that the former example actually describes a pointer to a character, not a string. See below for string lens, which is applicable to these cases.

LENSES

Lenses change the way that types are described. In the simplest case, a lens is directly a type. Otherwise a type is decorated by the lens. Ltrace understands the following lenses:

oct(TYPE)

The argument, which should be an integer type, is formatted in base-8.

hex(TYPE)

The argument, which should be an integer or floating point type, is format? ted in base-16. Floating point arguments are converted to double and then displayed using the %a fprintf modifier.

hide(TYPE)

The argument is not shown in argument list.

bool(TYPE)

Arguments with zero value are shown as "false", others are shown as "true".

bitvec(TYPE)

Underlying argument is interpreted as a bit vector and a summary of bits set in the vector is displayed. For example if bits 3,4,5 and 7 of the bit vec? tor are set, ltrace shows <3-5,7>. Empty bit vector is displayed as <>. If there are more bits set than unset, inverse is shown instead: e.g. ~<0> when a number 0xfffffffe is displayed. Full set is thus displayed ~<>.

If the underlying type is integral, then bits are shown in their natural big-endian order, with LSB being bit 0. E.g. bitvec(ushort) with value 0x0102 would be displayed as <1,8>, irrespective of underlying byte order. For other data types (notably structures and arrays), the underlying data is interpreted byte after byte. Bit 0 of first byte has number 0, bit 0 of second byte number 8, and so on. Thus bitvec(struct(int)) is endian sensi? tive, and will show bytes comprising the integer in their memory order. Pointers are first dereferenced, thus bitvec(array(char, 32)*) is actually a pointer to 256-bit bit vector.

string(TYPE)

string[EXPR]

string

The first form of the argument is canonical, the latter two are syntactic

sugar. In the canonical form, the function argument is formatted as string.

The TYPE shall be either a char*, or array(char,EXPR), or array(char,EXPR)*.

If an array is given, the length will typically be a zero expression (but doesn't have to be). Using argument that is plain array (i.e. not a pointer to array) makes sense e.g. in C structs, in cases like struct(string(array(char, 6))), which describes the C type struct {char s[6];}.

Because simple C-like strings are pretty common, there are two shorthand forms. The first shorthand form (with brackets) means the same as string(array(char, EXPR)*). Plain string without an argument is then taken to mean the same as string[zero].

Note that char* by itself describes a pointer to a char. Ltrace will dereference the pointer, and read and display the single character that it points to.

enum(NAME[=VALUE]{,NAME[=VALUE]})

enum[TYPE](NAME[=VALUE]{,NAME[=VALUE]})

This describes an enumeration lens. If an argument has any of the given values, it is instead shown as the corresponding NAME. If a VALUE is omitted, the next consecutive value following after the previous VALUE is taken instead. If the first VALUE is omitted, it's 0 by default.

TYPE, if given, is the underlying type. It is thus possible to create enums over shorts or longs?arguments that are themselves plain, non-enum types in C, but whose values can be meaningfully described as enumerations. If omitted, TYPE is taken to be int.

TYPE ALIASES

A line in config file can, instead of describing a prototype, create a type alias. Instead of writing the same enum or struct on many places (and possibly updating when it changes), one can introduce a name for such type, and later just use that name:

typedef NAME = LENS;

RECURSIVE STRUCTURES

Ltrace allows you to express recursive structures. Such structures are expanded to the depth described by the parameter -A. To declare a recursive type, you first have to introduce the type to ltrace by using forward declaration. Then you can

use the type in other type definitions in the usual way:

typedef NAME = struct;

typedef NAME = struct(NAME can be used here)

For example, consider the following singy-linked structure and a function that takes such list as an argument:

typedef int_list = struct;

typedef int_list = struct(int, int_list*);

void ll(int_list*);

Such declarations might lead to an output like the following:

ll({ 9, { 8, { 7, { 6, ... } } } }) = <void>

Ltrace detects recursion and will not expand already-expanded structures. Thus a doubly-linked list would look like the following:

typedef int_list = struct;

typedef int_list = struct(int, int_list*, int_list*);

With output e.g. like:

ll({ 9, { 8, { 7, { 6, ..., ... }, recurse^ }, recurse^ }, nil })

The "recurse^" tokens mean that given pointer points to a structure that was ex?panded in the previous layer. Simple "recurse" would mean that it points back to this object. E.g. "recurse^^^" means it points to a structure three layers up.

For doubly-linked list, the pointer to the previous element is of course the one that has been just expanded in the previous round, and therefore all of them are either recurse^, or nil. If the next and previous pointers are swapped, the output adjusts correspondingly:

ll({ 9, nil, { 8, recurse^, { 7, recurse^, { 6, ..., ... } } } })

EXPRESSIONS

Ltrace has support for some elementary expressions. Each expression can be either of the following:

NUM An integer number.

argNUM Value of NUM-th argument. The expression has the same value as the corre?sponding argument. arg1 refers to the first argument, arg0 to the return value of the given function.

retval Return value of function, same as arg0.

eltNUM Value of NUM-th element of the surrounding structure type. E.g.

struct(ulong,array(int,elt1)) describes a structure whose first element is a length, and second element an array of ints of that length.

zero

zero(EXPR)

Describes array which extends until the first element, whose each byte is 0. If an expression is given, that is the maximum length of the array. If NUL terminator is not found earlier, that's where the array ends.

PARAMETER PACKS

Sometimes the actual function prototype varies slightly depending on the exact pa? rameters given. For example, the number and types of printf parameters are not known in advance, but ltrace might be able to determine them in runtime. This fea? ture has wider applicability, but currently the only parameter pack that ltrace supports is printf-style format string itself:

format When format is seen in the parameter list, the underlying string argument is parsed, and GNU-style format specifiers are used to determine what the fol? lowing actual arguments are. E.g. if the format string is "%s %d\n", it's as if the format was replaced by string, string, int.

RETURN ARGUMENTS

C functions often use one or more arguments for returning values back to the caller. The caller provides a pointer to storage, which the called function ini? tializes. Ltrace has some support for this idiom.

When a traced binary hits a function call, ltrace first fetches all arguments. It then displays left portion of the argument list. Only when the function returns does ltrace display right portion as well. Typically, left portion takes up all the arguments, and right portion only contains return value. But ltrace allows you to configure where exactly to put the dividing line by means of a + operator placed in front of an argument:

int asprintf(+string*, format);

Here, the first argument to asprintf is denoted as return argument, which means that displaying the whole argument list is delayed until the function returns:

a.out->asprintf(<unfinished ...>

libc.so.6->malloc(100) = 0x245b010

[... more calls here ...]

<... asprintf resumed> "X=1", "X=%d", 1) = 5

It is currently not possible to have an "inout" argument that passes information in both directions.

EXAMPLES

In the following, the first is the C prototype, and following that is ltrace con? figuration line.

```
void func_charp_string(char str[]);
    void func_charp_string(string);
enum e_foo {RED, GREEN, BLUE};
void func_enum(enum e_foo bar);
    void func_enum(enum(RED,GREEN,BLUE));
        - or -
    typedef e_foo = enum(RED,GREEN,BLUE);
    void func_enum(e_foo);
void func_arrayi(int arr[], int len);
    void func_arrayi(array(int,arg2)*,int);
struct S1 {float f; char a; char b;};
struct S2 {char str[6]; float f;};
struct S1 func_struct(int a, struct S2, double d);
    struct(float,char,char)  func_struct_2(int,  struct(string(array(char,
    6)),float), double);
```

AUTHOR

Petr Machata <pmachata@redhat.com>