Generic helper / demonstration library

Generated by Doxygen 1.9.1

Chapter 1

GenericHelpers

This library exists to handle a wide array of things that I've found are fairly common to things I do on a day to day basis. I welcome any contributions to this library - I simply ask that they be written in such a way as to be as generic as possible, so they can be used.

All contributed code *MUST* be documented using DoxyGen typical format as this library is also created to help people learn and understand.

Because of the doxygen configurations building this requires having texlive and pdflatex installed (On Ubuntu texlive-latext-base texlive-latex-extra texlive-latex-recommended and pdflatex packages)

To build with ninja:

cmake -G Ninja . ninja

In docs/latex there is a Makefile, and running make in docs/latex after running ninja will generate a refman.pdf pdf reference manual. Note that this requires pdflatex to be installed

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 bgp_ipv4_prefix Struct Reference

`#include <decode_helpers.h>`

Data Fields

- __u32 prefix
- __u8 cidr
- __u8 * next

4.1.1 Field Documentation

4.1.1.1 cidr

`__u8 bgp_ipv4_prefix::cidr`

4.1.1.2 next

`__u8* bgp_ipv4_prefix::next`

The cidr of the prefix Pointer to the next entry NLRI entry

4.1.1.3 prefix

`__u32 bgp_ipv4_prefix::prefix`

The network byte order prefix read from a BGP NLRI

The documentation for this struct was generated from the following file:

- decode_helpers.h

4.2 proto_msg Struct Reference

`#include <decode_helpers.h>`

Data Fields

- __u64 result
- __u64 msg_code
- __u64 msg_type

4.2.1 Field Documentation

4.2.1.1 msg_code

`__u64 proto_msg::msg_code`

The message code derived from the result

4.2.1.2 msg_type

`__u64 proto_msg::msg_type`

The message type derived from the result

4.2.1.3 result

`__u64 proto_msg::result`

The 64 bit variable integer decode

The documentation for this struct was generated from the following file:

- decode_helpers.h

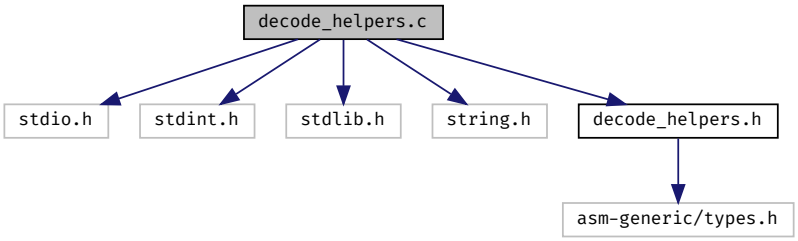
Chapter 5

File Documentation

5.1 decode_helpers.c File Reference

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "decode_helpers.h"
```

Include dependency graph for decode_helpers.c:



Functions

- void dump_buffer (void *buffer, __u16 size)

  dump_buffer takes a pointer and dumps a specific number of bytes
- void reverse_array_10 (__u8 bytes[ ])

  This function reverses a 10 byte array.
- struct proto_msg * mem_to_msg (const __u8 *ptr)

  This function is used to decode protobuf messages and message types.
- struct bgp_ipv4_prefix * read_bgp_prefix (__u8 *ptr)

  Reads a BGP encoded prefix from an NLRI or withdraw message.
- __u8 get_var_int (const u_char *data, __u64 *varint)

  this function gets a variable encoded 64 bit integer from a 7 byte encoded string of bytes.

5.1.1 Function Documentation

5.1.1.1 dump_buffer()

```
void dump_buffer (
            void * buffer,
            __u16 size )
```

dump_buffer takes a pointer and dumps a specific number of bytes

This function dumps memory in a format that is importable by wireshark or other programs that can import hex dumps

Parameters

in	*buffer*	A pointer to the memory to be dumped
in	*size*	The number of bytes to dump

5.1.1.2 get_var_int()

```
__u8 get_var_int (
            const u_char * data,
            __u64 * varint )
```

this function gets a variable encoded 64 bit integer from a 7 byte encoded string of bytes.

Parameters

in	*data*	A pointer to the bytes we are extracting the variable integer from
in	*varint*	A pointer to where the extracted varint will be stored

Returns

The number of bytes used to encode the extracted varint

Zero out the varint before we start

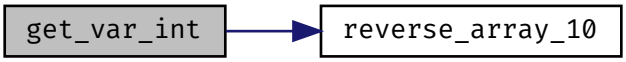
```
*varint = 0
```

If the first byte does not have the high order bit set take that as a standalone byte, else transverse the memory pointed to by data until either we have 10 bytes or until the high order bit is not set

```
if ((data[0] & 128) == 128) {
  bytes[length] = data[data_count];
  byte_count = 1;
} else {
  while ((data[data_count] & 128) == 128) {
      bytes[length--] = data[data_count++];
      byte_count++;
      if (byte_count == 9)
          break;
```

```
  }
  bytes[length] = data[data_count];
}
```

Here is the call graph for this function:



5.1.1.3 mem_to_msg()

```
struct proto_msg* mem_to_msg (
            const __u8 * ptr )
```

This function is used to decode protobuf messages and message types.

This function reads a maximum of 10 bytes of memory into an array, terminating when one byte does not have its high order bit set. It then reverses the array, and concatenates the byte array into a single 64 bit integer, using only the low order 7 bits of each byte in the array. The result is then shifted right by 3 to produce the message code and AND'd by 7 to get the high order bytes to find the message type NOTE: It is important that this function returns allocated memory, and the result must be free'd by the calling function to avoid memory leaks.

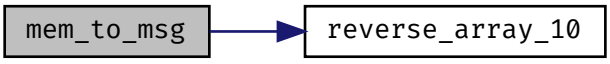
Parameters

in	*ptr*	A pointer to the memory containing the bytes requiring decode

Returns

An allocated proto_msg structure or NULL if allocation of memory fails

Here is the call graph for this function:



5.1.1.4 read_bgp_prefix()

```
struct bgp_ipv4_prefix* read_bgp_prefix (
            __u8 * ptr )
```

Reads a BGP encoded prefix from an NLRI or withdraw message.

See also

bgp_ipv4_prefix @detail This function decodes compressed ipv4 prefix's as contained in bgp update messages. The first byte in the message represents the CIDR mask (The number of bits in the prefix). If the CIDR is fully divisible by 8, then CIDR/8 bytes are copied into the prefix element. If CIDR is NOT fully divisible by 8 then (CIDR/8)+1 bytes are copied into the prefix element. The next pointer element is set using the input pointer + 1 byte for the CIDR and then X bytes for the prefix itself, where X is either CIDR/8 or (CIDR/8)+1 NOTE: This function returns a pointer that must be free'd by the caller to avoid memory leaks. This function only serves for demonstration purposes because this type of processing would normally be done inline in a BGP update function without the need of additional memory allocation.

Parameters

in	*ptr*	A Pointer to the start of the compressed prefix

5.1.1.5 reverse_array_10()

```
void reverse_array_10 (
            __u8 bytes[] )
```

This function reverses a 10 byte array.

This function is used primarily for handling variable integer decodes VARINT's are something used heavily in gbp file formats (Google protobuffers) NOTE: This function assumes that the input array is at least 10 bytes long and should the input array be less than 10 bytes this may cause unexpected behavior

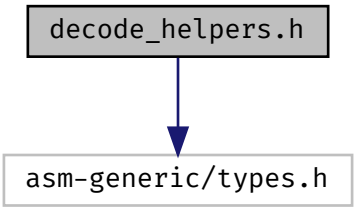
Parameters

in	*bytes*	An array of bytes to be reversed.

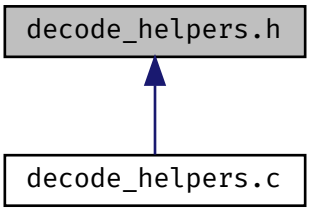
5.2 decode_helpers.h File Reference

```
#include <asm-generic/types.h>
```

Include dependency graph for decode_helpers.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct bgp_ipv4_prefix
- struct proto_msg

Functions

- void dump_buffer (void *buffer, __u16 size)

  dump_buffer takes a pointer and dumps a specific number of bytes
- void reverse_array_10 (__u8 bytes[ ])

  This function reverses a 10 byte array.
- struct proto_msg * mem_to_msg (const __u8 *ptr)

  This function is used to decode protobuf messages and message types.
- struct bgp_ipv4_prefix * read_bgp_prefix (__u8 *ptr)

  Reads a BGP encoded prefix from an NLRI or withdraw message.
- __u8 get_var_int (const u_char *data, __u64 *varint)

  this function gets a variable encoded 64 bit integer from a 7 byte encoded string of bytes.

5.2.1 Function Documentation

5.2.1.1 dump_buffer()

```
void dump_buffer (
            void * buffer,
            __u16 size )
```

dump_buffer takes a pointer and dumps a specific number of bytes

This function dumps memory in a format that is importable by wireshark or other programs that can import hex dumps

Parameters

in	*buffer*	A pointer to the memory to be dumped
in	*size*	The number of bytes to dump

5.2.1.2 get_var_int()

```
__u8 get_var_int (
            const u_char * data,
            __u64 * varint )
```

this function gets a variable encoded 64 bit integer from a 7 byte encoded string of bytes.

Parameters

in	*data*	A pointer to the bytes we are extracting the variable integer from
in	*varint*	A pointer to where the extracted varint will be stored

Returns

The number of bytes used to encode the extracted varint

Zero out the varint before we start

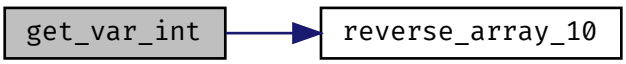
```
*varint = 0
```

If the first byte does not have the high order bit set take that as a standalone byte, else transverse the memory pointed to by data until either we have 10 bytes or until the high order bit is not set

```
if ((data[0] & 128) == 128) {
  bytes[length] = data[data_count];
  byte_count = 1;
} else {
  while ((data[data_count] & 128) == 128) {
      bytes[length--] = data[data_count++];
      byte_count++;
      if (byte_count == 9)
          break;
```

```
  }
  bytes[length] = data[data_count];
}
```

Here is the call graph for this function:



5.2.1.3 mem_to_msg()

```
struct proto_msg* mem_to_msg (
            const __u8 * ptr )
```

This function is used to decode protobuf messages and message types.

This function reads a maximum of 10 bytes of memory into an array, terminating when one byte does not have its high order bit set. It then reverses the array, and concatenates the byte array into a single 64 bit integer, using only the low order 7 bits of each byte in the array. The result is then shifted right by 3 to produce the message code and AND'd by 7 to get the high order bytes to find the message type NOTE: It is important that this function returns allocated memory, and the result must be free'd by the calling function to avoid memory leaks.

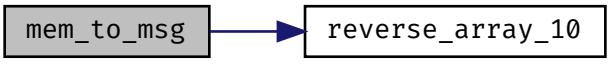
Parameters

in	*ptr*	A pointer to the memory containing the bytes requiring decode

Returns

An allocated proto_msg structure or NULL if allocation of memory fails

Here is the call graph for this function:



5.2.1.4 read_bgp_prefix()

```
struct bgp_ipv4_prefix* read_bgp_prefix (
            __u8 * ptr )
```

Reads a BGP encoded prefix from an NLRI or withdraw message.

See also

bgp_ipv4_prefix @detail This function decodes compressed ipv4 prefix's as contained in bgp update messages. The first byte in the message represents the CIDR mask (The number of bits in the prefix). If the CIDR is fully divisible by 8, then CIDR/8 bytes are copied into the prefix element. If CIDR is NOT fully divisible by 8 then (CIDR/8)+1 bytes are copied into the prefix element. The next pointer element is set using the input pointer + 1 byte for the CIDR and then X bytes for the prefix itself, where X is either CIDR/8 or (CIDR/8)+1 NOTE: This function returns a pointer that must be free'd by the caller to avoid memory leaks. This function only serves for demonstration purposes because this type of processing would normally be done inline in a BGP update function without the need of additional memory allocation.

Parameters

in	*ptr*	A Pointer to the start of the compressed prefix

5.2.1.5 reverse_array_10()

```
void reverse_array_10 (
            __u8 bytes[] )
```

This function reverses a 10 byte array.

This function is used primarily for handling variable integer decodes VARINT's are something used heavily in gbp file formats (Google protobuffers) NOTE: This function assumes that the input array is at least 10 bytes long and should the input array be less than 10 bytes this may cause unexpected behavior

Parameters

in	*bytes*	An array of bytes to be reversed.

5.3 README.md File Reference

Index