The USC 2.0 Reference Manual[1]

Sean O'Malley, Todd Proebsting, Gregg Townsend, H. Dan Lambright
University of Arizona[2]

TR 95-12

Abstract

This document is the reference manual for version 2.0 of the Universal Stub Compiler (USC) and the USC Inference Tool (USIT). USC 2.0 is a nearly complete re-implementation with some changes to the syntax of the USC language and the USC and USIT program interface. USC 2.0 is not backwardly compatible with the previous version of USC. USC is a stub compiler that generates stubs that perform many data conversion operations. USC is flexible and can be used in situations where previously only manual code generation was possible. USC generated code is up to 20 times faster than code generated by traditional argument marshaling schemes such as ASN.1 and Sun XDR. USIT is a small tool that takes C typedefs and generates those typedefs with USC annotations for the native byte order and alignment for the compiler and host architecture it is run on.

November 2, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1] This work was supported by ARPA under the grant DABT63-91-C-0030.

[2] Authors' addresses: Department of Computer Science, The University of Arizona, Tucson, AZ 85721. Email: {sean,todd,gmt,hdlambri}@cs.arizona.edu.

1 Introduction

This document is the reference manual for version 2.0 of the Universal Stub Compiler (USC) and the USC Inference Tool (USIT). USC 2.0 is a nearly complete re-implementation with some changes to the syntax of the USC language and the USC and USIT program interface. USC 2.0 is not backwardly compatible with the previous version of USC. Users that have used the previous version of USC should probably start by reading the differences section below.

USC is a stub compiler that generates stubs that perform many data conversion operations. USC is flexible and can be used in situations where previously only manual code generation was possible. USIT is a small tool that takes a file of C typedefs and generates a program which figures out the native byte-order and alignment of the types defined in the file. USIT uses this information to produce a file containing the original typedefs with the appropriate USC annotations. This file can then be included in a USC program by running the C preprocessor over the USC file before USC is run. More information about the design USC can be found in [O'Malley].

The syntax of a USC program is a subset of the ANSI C syntax extended to allow the user to annotate data type definitions with byte order and alignment information. With this syntax the user declares type definitions and functions that manipulate values of these types. With minor exceptions a USC program stripped of its annotations is a valid C program. Below is the USC program tcp.usc.

```
%%
/* define native DECstation base types */
typedef (4,4,<0..3>) long;
typedef (4,4,<0..3>) int;
typedef (2,2,<0..1>) short;
typedef (1,1,<0>) char;

/* tcp header in native DECstation format */
typedef struct tcp_native_hdr {
  unsigned short   sport(2,0,<0..1>),
                   dport(2,2,<0..1>);
  unsigned int     x2(1,4,<0>):4 <4..7>,
                   off(1,4,<0>):4 <0..3>;
  unsigned long    seq(4,8,<0..3>),
                   ack(4,12,<0..3>);
  unsigned char    flags(1,16,<0>);
  unsigned short   win(2,18,<0..1>),
                   sum(2,20,<0..1>),
                   urp(2,22,<0..1>);
} native_hdr(22,4,0);

/* tcp header in network format */
typedef struct tcp_net_hdr {
  unsigned short   sport(2,0,<1..0>),
                   dport(2,2,<1..0>);
  unsigned int     x2(1,4,<0>):4 <4..7>,
                   off(1,4,<0>):4 <0..3>;
  unsigned long    seq(4,5,<3..0>),
                   ack(4,9,<3..0>);
  unsigned char    flags(1,13,<0>);
```

```
  unsigned short   win(2,14,<1..0>),
                   sum(2,16,<1..0>),
                   urp(2,18,<1..0>);
} net_hdr(20,1,0);

/* a stub which converts from host to network form */
void tcphdr(net_hdr *src, native_hdr *dest)
{
  *dest = *src;
}
%%
```

In the example above the first four lines define the native format of all base types for the compiler/machine combination for which USC will be generating code (in this case GCC compiling for a DECstation 5000). The USC program then goes on to define two types: net_hdr and native_hdr. The type net_hdr is a variant of the TCP header in big endian byte order and is packed into minimal space without regard to the alignment of any of its fields. The type native_hdr is the same variant of the TCP header in little endian byte order with the structure padded so that each field is aligned appropriately. The stub tcphdr takes a TCP header in network format and copies it to a TCP header in DECstation 5000 format. From this program USC can automatically generate a C function that performs the correct reformatting operations for the compiler and host defined in the USC program.

2 USC File Formats

The USC input and output file formats are somewhat similar to those used by Yacc. A USC input file looks as follows:

```
<arbitrary text>
%%
<USC program>
%%
<more arbitrary text>
```

From a given input file USC generates the following output file:

```
<arbitrary text>
/* %% */
<USC generated C code>
/* %% */
<more arbitrary text>
```

The USC generated C code consists of C macro implementations of all USC stubs defined with the macro keyword and C function implementations of all remaining USC stubs. As an option (-p filename) the user may request that USC generate a separate file containing the C macro implementations of all USC stubs defined with the macro keyword. The file will also contain ANSI prototypes for all of the C functions defined in the main USC output file. All C functions generated by USC as well as the arbitrary text at the beginning and end of the input file will still appear in the main output file.

Note that USC makes no attempt to generate C typedefs corresponding to the USC typedefs found in the USC program. In standard mode USC generates functions with void * arguments in place of all pointers to USC defined types. Optionally (-t) the user may have USC generate functions whose arguments types are identical to those found in the USC stub definitions. To be compiled the user must write C typedefs for each USC typedef used as an argument to a stub. For native types the appropriate C typedef is the USC typedef minus the annotations. For non-native types the user can define a structure which consists of a single array of characters as long as the total length of the USC type. See the USC man page for more information.

3 USC Language Description

3.1 USC Type System

The USC type system is simple and supports the type void and the base types char, short, int, and long (long long and enumerated types are not supported). All integral types, including char and bit-fields, are assumed to be signed unless declared unsigned. USC does not support auto, register, static, const, or volatile types. In addition, USC supports structures, unions, and arrays of these types, as well as bit-fields. Pointer types are only allowed in stub parameter and return value declarations. Pointer types are used to pass data by reference and to return values of the address of operation. The typedef operation is supported. USC does not support floating point types or arbitrary pointer-based objects. Unions are only partially supported. USC allows unions to be defined, and a union's fields to be selected, but union assignment is not supported. Since C supports un-tagged unions there is no information indicating which union field is currently active and thus USC cannot predict which field to convert. Furthermore, USC must assume that the length of a union is determined by its largest field. In most network representation the length of a union is determined by the field which is currently active. Thus the USC union type cannot be used to generate stubs for XDR unions which are variable length.

3.2 USC Data Layout Annotations

USC provides a notation for precisely defining the layout of each variable passed to a USC stub. USC makes no assumptions about the byte order of any type. The input file must precisely specify the correct byte order and offset of every type. Typedefs are also used to inform USC of the native format of the integral base types on the compiler/host combination that will be used to compile and execute the generated stub. Thus USC requires the user to define three potentially different formats: the format of the source of any assignment, the format of the destination of any assignment, and the format of the C integral types that will be used to perform the assignment.

USC supports annotations in several places in USC typedefs: before a native type, after the a type name, after the field name in a structure or union, and after the size of a bit field. The meaning of these annotations can differ based upon where in the typedef the annotation appears and what type is being defined. For example let us take the simplest case: the USC annotation found after the type definition of a simple base type.

```
typedef type name(size, alignment, byte order);
```

or more concretely:

```
typedef int foo(4, 4, <0,1,2,3>);
```

The first field in this annotation defines the size of the type; in this case foo is a 4 byte int. The second field defines that type's minimum alignment; in this case a foo must be 4-byte aligned. The alignment field is a guarantee to the compiler that the address of the annotated variable modulo alignment will be equal to zero. An alignment of 1 will always generate correct code. In general the higher the alignment specified the better the code USC will generate. It is possible to specify an alignment for a type that is more restrictive than the alignment used by the compiler. The third field is used to specify which memory bytes, in what order, are used to represent a given type; in this case the byte order is little endian. This annotation is actually a simplification of the more complex annotation give below.

```
typedef type name(Tsize[/Msize], alignment, byte order);
```

or more concretely:

```
typedef int foo(4/4, 4, <0,1,2,3>);
```

Because the size of the type (the Tsize) may not be equal to the size of the memory region the type is stored in (the Msize) there must be some way of specifying this. Thus the USC syntax gives the user the option of specifying the actual Msize with the syntax given above. If you had a four byte integer stored in the first 4 bytes of a little endian 8 byte word you would use the following USC annotation:

```
typedef int foo(4/8, 8, <0,1,2,3>);
```

The syntax of the byte order field depends upon the value of the Tsize and Msize. Byte order is defined by a comma separated list of Tsize distinct integers between 0 and Msize-1 enclosed in angle brackets ($\langle$1,2,3$\rangle$). A range may be used to abbreviate a list of integers. A range has the form n..m and is equivalent to the list n, n+1, ... m if m $>$ n. If n $>$ m the range n..m is equivalent to the list n, n-1, ... m. This list is interpreted as a transformation from the byte number of the variable to the offset of that byte from the start of the variable in memory. Thus in USC big-endian byte orders are represented by a decreasing sequence of numbers while little-endian byte orders are represented by increasing sequences.

The annotations that appear before native integral types are interpreted in the same way as annotations that appear after the type name of a base type. These annotations define the Msize, Tsize, alignment, and byte order of the native integral type so annotated. The annotation below tells USC that the native type int of the compiler/machine the USC generated C program will be compiled and run on is 4 bytes long, 4 byte aligned and little endian.

```
typedef (4, 4, <0,1,2,3>) int;
```

When the type being annotated is a structure the annotation of the type name is interpreted differently. Both the Tsize and the byte-order have already been defined by the annotations of the structure field names. If the Tsize/Msize notation is used to define the length of the overall structure the Tsize is ignored except to check to see that it is less than or equal to the Msize. If only a single length is given it is interpreted as the Msize and the Tsize is simply ignored. The byte-order field must be zero.

When annotating a field of a structure (as opposed to the structure iself) USC must know the offset of that field from the beginning of the structure. Once the offset is known the alignment of a field can be computed from the offset and the alignment of the structure as a whole. Therefore in the USC annotation of the field of a structure the second field in the annotation is defined as the offset from the beginning of the structure. The size and byte order annotations are unchanged.

Note that any offset is legal. USC places no restrictions on the order that fields in any structure. Overlapping field definitions are allowed. The annotations used to define structures and fields in structures are given below:

```
typedef struct {
    base_type name(Tsize[/Msize], offset, byte order);
} name(Msize, alignment, 0);
```

or more concretely:

```
typedef struct {
    unsigned char a(1,0,<0>);
    unsigned char b(1,1,<0>);
    unsigned char c(1,2,<0>);
    unsigned char d(1,3,<0>);
} IPHost(4, 4, 0);
```

Important note. USC uses structural type compatibility and basically ignores the the field name. Thus when a USC stub is generated the first field defined in one structure is copied to the first field defined in the other structure regardless of the names of the fields. Thus assignment is based upon lexical order. Once you define a set of fields in a given order you can change where those fields appear in the memory in an arbitrary way using USC offset annotation. However USC will still copy the first field in one structure definition to first field defined in the other whatever there relative physical location.

Bit-fields can be further annotated after the size of the bit-field with the bit-order of the bit-field. This is defined with exactly the same syntax as used in defining byte-orders but the size of the bit-field in bits is used as the Tsize and the sizeof of the containing base type in bits is used as the Msize. Both of these numbers can be calculated from numbers found in the C bit-field notation or the USC annotation of the containing integral type these numbers. The offset of the containing integral type combined with the bit order specification is all that is required to generate code. For example the annotation describes the bit order of two bit-fields found in the header given above.

```
u_int  x2  (1,4,<0>):4<4..7>,
       off (1,4,<0>):4<0..3>;
```

The annotation used to describe any array type is the same as would be used to describe the base type that makes up the arrays. The reason for this is that USC assumes all arrays are laid out as they are in C. That is row major order. As in C, USC does no array bounds checking when arrays are accessed. An array of 10 shorts where each short is stored in the last two bytes of a word could be described as follows:

```
short a(2/4,4,<2,3>)[10];
```

Note that type annotations are unrelated to the host for which USC generates stubs. Thus it is possible to generate stubs for an Intel x86 which converts a type in native DEC C, VAX format to native GCC SPARC format. The only host dependency in USC is that USC must be run on a host whose long int is at least as large as any declared native type of the target. So you would not be able to generate the code for a USC program on the Sun and use it on the Alpha unless you omit the definition of the Alpha's native 8-bit long type to make USC generate code using only ints.

3.3 Redefining the Annotation of a Type

USC allows you some limited flexibility to redefine the layout of a type. Suppose you wanted to define a new type identical to tcp header in network format defined above except that it is four byte aligned.

```
typedef  net_hdr net_hdr_4byte(20,4,0);
```

The above code defines a data type net_hdr_4byte with the layout specified in net_hdr except that the alignment is 4 rather than 1 as it was the net_hdr definition. The annotations of the fields net_hdr_4byte structure are assumed to be identical to the annotations of the fields of the net_hdr structure. Note that unlike USC 1.0 you must provide the correct value for each field of the annotation.

3.4 Annotating Native Integral Types

Users should annotate every integral base type in every USC program. Even if a base type is not used in the USC source the USC code generator may want to generate code using that base type to improve efficiency. If no base type annotations are given the generated code will only use character moves.

Unfortunately there is no way in the current USC release to tell the code generator to not use characters. This feature would be useful when writing device drivers whose memory mapped memory buffers support only word reads and writes.

3.5 USC Scope Rules

USC's scope rules are similar to C, except that names of typedefs or functions may not be reused in another context or inner scope.

3.6 USC Type Compatibility

The introduction of data layout annotations introduces three distinct levels of type compatibility into USC. Two USC types are type compatible if their underlying ANSI C types are structurally compatible. Two USC types are copy compatible if they are type compatible assuming all integral types to be identical. Two USC types are identical if they are type compatible and have identical annotations.

3.7 USC Stub definitions

Stubs are defined in USC as functions are defined in ANSI C except that USC does not support varags. Note that contrary to the USC paper only ANSI style parameter lists are supported. The user defines the parameters and return value to a stub exactly as they would in a C function except that USC's type system is used. The body of a USC stub is defined using a restricted subset of the ANSI C statement grammar. Only assignment and return statements are supported. Statements are defined using a restricted subset of the C expression grammar. The key feature of this expression grammar is that it works on annotated USC types. Assignments will correctly convert values when assigning between two structures with different layouts.

In the USC expression grammar component selection (`->` and `.`), array subscripting(`[]`), indirection (`*`), sizeof and address of (`&`) are supported on all appropriate types. The assignment operation is supported between all copy compatible USC types. The type of array indices and the type of the operands of the operations addition(+), subtraction(−), multiplication(`*`) and division(`/`) must be identical to one of the native base types given in the typedefs at the beginning of the USC program. Expressions and constants are only allowed in array subscripts or on the right side of assignment or return statements.

Parameters to USC stubs must be either pointers to any USC type, or a type that is copy compatible to a native base type. The value returned by a USC stub must be type void, a pointer to any USC type, or a type copy compatible to a native base type. Thus, USC stubs can take as parameters or return base types in any byte order. The code below copies one one array element (in this case an integer) from a user specified location in one array to a user specified location in another.

```
typedef (4,4,<0,1,2,3>) int;
typedef (2,2,<0,1>) short;
typedef (1,1,<0>) char;

typedef int array (4,4,<0,1,2,3>) [100];


void foo (int len1, int len2, array a1, array a2)
{
  a1[len1] = a2[len2];
}
```

USC supports the standard C static qualifier for functions. A USC stub defined as static will result in the generated C function being defined as static. USC supports the inline qualifier found in many C compilers. A USC stub declared as inline will generate an inline function. In addition, USC supports the qualifier macro which directs USC to produce a C macro implementation of the specified stub. The qualifier macro may only be used on stubs returning void.

The stub tcphdr defined in section 1 shows how to define a stub to copy a TCP header from network format to DECstation 5000 native format. The generated C code swaps bytes and realigns the data. Less traditional stubs can also be generated. It is often useful to read and write fields into a network header stored in network format. Below is stub that peeks into a TCP header in network format and returns the offset field as an native int.

```
int tcpgetoff (net_hdr *hdr)
{
  return hdr->off;
}
```

3.8 In-Place Data Modification

Currently USC does not support the in place modification of data types. The USC code generator assumes that any two pointers passed to a USC stub are not aliased.

4 USIT: The USC Inference Tool

The correctness of a USC stub is dependent upon the accuracy of the data layout annotations. For headers in network format this is generally not a problem because the precise data layout of the header is included in the standard and once a USC type has been defined for that layout it can be used on all hosts and compilers. Getting the correct layout of the native compiler format of a network header is another matter. It is rarely specified by the compiler documentation and it changes for each host/compiler pair. Annotating such types manually could be as error prone and time consuming as writing byte swapping code by hand.

To eliminate this problem we have written the USC Inference Tool (USIT) to determine the alignment and byte order of native variables. USIT takes a file containing valid a list of valid C typedefs (without any USC annotations) and produces a list of valid USC typedefs annotated correctly for the native format of host and compiler USIT was executed on. This list of typedefs can then be included in a USC program using the C pre-processor on most hosts. Note the input to USIT is not an arbitrary C .h file. Only typedefs are allowed. Variable declarations of any type will be flagged as syntax errors. The output of USIT is not a valid USC program. It is simply a list of USC annotated typedefs.

USIT supports two basic modes of operation: single stage and two stage mode. In single stage mode USIT takes in a usc file with some number of un-annotated typedefs and generates a fully annotated usc file in a single stage. While options permit the user to select the C compiler used when computing the native representation the host used will be the one on which USIT was run on. For example:

```
usit -o udp.usc udp.usit
```

generates the fully annotated usc file udp.usc as output. In two stage mode USIT is executed twice. The first execution produces a C program which can be compiled and run on hosts that do not support USIT. Executing the USIT generated C program produces a data file that is then passed to the second execution of USIT. This causes USIT to generate a USC file annotated with the representation of the compiler the C program was compiled with and the host the C program was executed on. The following is equivalent to the single stage USIT example above:

```
usit -i -o test.c udp.usit
gcc -O test.c
```

```
a.out > test.data
usit -d test.data -o udp.usc udp.usit
```

USIT can also be used to generate the annotations for the base types of a compiler using the -n option.

4.1 USC, USIT, and the C Preprocessor

Both USC and USIT can take the output of the C preprocessor as input. The primary intended use of CPP in USC programs is to include files generated by USIT. The primary intended use of CPP in USIT programs is to fully resolve all CPP defined integer constants used in array definitions. It is probably possible to use other CPP functions such as selective compilation (ifdefs) on USIT and USC files. However, such uses have not been tested.

Both USC and USIT consume any preprocessor generated line number information and do not pass it to their output files. USC does however generate ANSI line number commands which allow syntax checkers to point to the offending lines and files. (This is not true in USIT). Note that not all CPP's are compatible with USC: For example, Sun's version of CPP objects to USC's byte order notation.

5 Extended Example: RIP

The following example demonstrates the use of USC/USIT to generate a portable implementation of the header marshaling routines for the RIP protocol. All the necessary code to perform byte-swapping for the RIP packet header is generated by USC, and USIT is used to annotate the native types. This program consists of three files: rip_const.h, rip.h, and rip.usc. rip_const.h contains the sizes of the arrays contained in the RIP packet format. Note that this file is included in both rip.h and rip.usc. The fact that rip_const.h is included in rip.h and rip.h is included in rip.usc does not imply that rip.usc will see the defines contained in rip_const.h. The include in rip.h is removed by running CPP before running USIT and never makes it to rip.usc. Below are the contents of rip_const.h:

```
#define RIPNULLADDR 8
#define MAXRIPROUTES    25      /* MAX routes per packet                */
```

rip.h contains the C typedefs of the types required to define a RIP packet. rip.h is given below:

```
#include "rip_const.h"

typedef struct iphost {
    unsigned char      a,b,c,d;
} IPhost;


typedef struct  riprt {
        short   rr_family;      /* 4BSD Address Family                  */
        short   rr_mbz;         /* must be zero                         */
        IPhost  rr_addr;        /* ip address                           */
        char    rr_mbz2[RIPNULLADDR];   /* must be zero                 */
        int     rr_metric;      /* distance (hop count) metric          */
} NativeRipRoute;

/* RIP packet structure */
```

```
typedef struct  rip {
        char    rip_cmd;        /* RIP command                              */
        char    rip_vers;       /* RIP_VERSION, above                       */
        short   rip_mbz;        /* must be zero                             */
        NativeRipRoute   rip_rts[MAXRIPROUTES];
} NativeRipPacket;
```

The defines found in rip_const.h and the typedefs found in rip.h were extracted from a number of existing C include files for the RIP protocol definition. This is necessary because USIT cannot handle arbitrary C include files. Thus the typedefs and defines of interest must be extracted from the existing include files and isolated in files which can then be processed by USIT. These new files can then be included in the .h files from which the typedefs and defines they contain was extracted. It would be much simpler if USIT could accept arbitrary .h files and ignore everything but the typedefs.

CPP is first run on rip.h to do the include and replace any defined constants. Then rip.h is run through USIT with the -n option. USIT annotates the typedefs found rip.h and prepends the USC annotations describing the native representation to the its output file; in this case rip.usit. The generated file rip.usit is given below:

```
/***********************************************************/
/*                                                         */
/*                 USIT Output                             */
/*                 10:35:51 AM Oct 23, 1995                */
/*                 File: rip.h.cpp                         */
/*                                                         */
/***********************************************************/

typedef (1, 1, <0>) char;
typedef (2, 2, <0,1>) short;
typedef (4, 4, <0..3>) int;
typedef (8, 8, <0..7>) long;

typedef struct {
  signed char rip_cmd(1, 0, <0>);
  signed char rip_vers(1, 1, <0>);
  signed short rip_mbz(2, 2, <0,1>);
  struct {
    signed short rr_family(2, 0, <0,1>);
    signed short rr_mbz(2, 2, <0,1>);
    struct {
      unsigned char a(1, 0, <0>);
      unsigned char b(1, 1, <0>);
      unsigned char c(1, 2, <0>);
      unsigned char d(1, 3, <0>);
    } rr_addr(4, 4, 0);
    signed char rr_mbz2(1, 8, <0>)[8];
    signed int rr_metric(4, 16, <0..3>);
  } rip_rts(20, 4, 0)[25];
} NativeRipPacket(504, 4, 0);
```

```
typedef struct {
  signed short rr_family(2, 0, <0,1>);
  signed short rr_mbz(2, 2, <0,1>);
  struct {
    unsigned char a(1, 0, <0>);
    unsigned char b(1, 1, <0>);
    unsigned char c(1, 2, <0>);
    unsigned char d(1, 3, <0>);
  } rr_addr(4, 4, 0);
  signed char rr_mbz2(1, 8, <0>)[8];
  signed int rr_metric(4, 16, <0..3>);
} NativeRipRoute(20, 4, 0);

typedef struct {
  unsigned char a(1, 0, <0>);
  unsigned char b(1, 1, <0>);
  unsigned char c(1, 2, <0>);
  unsigned char d(1, 3, <0>);
} IPhost(4, 1, 0);
```

Note that USIT fully instantiates each type (eliminating dependencies between types) and inverts the order that the types are output. Note also that all cpp additions have been removed.

rip.usc contains the USC and C code necessary to read and write RIP packets in network format. This isolates the RIP program from any dependency on the native format of the local compiler and host. The full text of rip.usc is given below:

```
/*
 * rip_hdr.usc
 * isolate the RIP code from the network format
 */

%%

/* constants needed to define a RIP packet */
#include "rip_const.h"

/* Native RIP packet structure: local to this host */

#include "rip.usit"

/* Network RIP packet structure:  defined by RFC-1058 */

typedef struct {
      unsigned char a(1, 0, <0>);
```

```
      unsigned char b(1, 1, <0>);
      unsigned char c(1, 2, <0>);
      unsigned char d(1, 3, <0>);
} NetIPhost(1, 4, 0);

typedef  struct {
    signed short rr_family(2, 0, <1,0>);
    signed short rr_mbz(2, 2, <1,0>);
    struct {
      unsigned char a(1, 0, <0>);
      unsigned char b(1, 1, <0>);
      unsigned char c(1, 2, <0>);
      unsigned char d(1, 3, <0>);
    } rr_addr(4, 4, 0);
    signed char rr_mbz2(1, 1, <0>)[RIPNULLADDR];
    signed int rr_metric(4, 16, <3..0>);
} NetRipRoute(20, 4, 0);

typedef struct {
  signed char rip_cmd(1, 0, <0>);
  signed char rip_vers(1, 1, <0>);
  signed short rip_mbz(2, 2, <1,0>);
  struct {
    signed short rr_family(2, 0, <1,0>);
    signed short rr_mbz(2, 2, <1,0>);
    struct {
      unsigned char a(1, 0, <0>);
      unsigned char b(1, 1, <0>);
      unsigned char c(1, 2, <0>);
      unsigned char d(1, 3, <0>);
    } rr_addr(4, 4, 0);
    signed char rr_mbz2(1, 1, <0>)[8];
    signed int rr_metric(4, 16, <3..0>);
  } rip_rts(20, 4, 0)[25];
} NetRipPacket(24, 4, 0);

/* redefine the network rip packet to be one byte aligned */
typedef NetRipPacket UNetRipPacket(24,1,0);

/* copy the rip fixed part of the packet into native format */
void ripFixCopyIn(NativeRipPacket *native_packet,
                  NetRipPacket *net_packet)
{
native_packet->rip_cmd = net_packet->rip_cmd;
native_packet->rip_vers = net_packet->rip_vers;
native_packet->rip_mbz = net_packet->rip_vers;
```

```
}

/* copy a rip route list into native format */
void ripRtCopyIn(NativeRipPacket *native_packet,
 NetRipPacket *net_packet,
                 int i)
{
native_packet->rip_rts[i] = net_packet->rip_rts[i];
}

/* copy the rip fixed part of the packet into network format */
void ripFixCopyOut(NativeRipPacket *native_packet,
                   NetRipPacket *net_packet)
{
        net_packet->rip_cmd = native_packet->rip_cmd;
        net_packet->rip_vers = native_packet->rip_vers;
        net_packet->rip_mbz = native_packet->rip_vers;
}

/* copy a rip route list into network format */
void ripRtCopyOut(NativeRipPacket *net_packet,
                  NetRipPacket *native_packet,
                  int i)
{
        net_packet->rip_rts[i] = native_packet->rip_rts[i];
}


/* the same as above only assume network packet is unaligned */

void ripFixUCopyIn(NativeRipPacket *native_packet,
                   UNetRipPacket *net_packet)
{
        native_packet->rip_cmd = net_packet->rip_cmd;
        native_packet->rip_vers = net_packet->rip_vers;
        native_packet->rip_mbz = net_packet->rip_vers;
}

/* copy a rip route list into native format */
void ripRtUCopyIn(NativeRipPacket *native_packet,
                  UNetRipPacket *net_packet,
                  int i)
{
        native_packet->rip_rts[i] = net_packet->rip_rts[i];
}
```

```
/* copy the rip fixed part of the packet into network format */
void ripFixUCopyOut(NativeRipPacket *native_packet,
                    UNetRipPacket *net_packet)
{
        net_packet->rip_cmd = native_packet->rip_cmd;
        net_packet->rip_vers = native_packet->rip_vers;
        net_packet->rip_mbz = native_packet->rip_vers;
}

/* copy a rip route list into network format */
void ripRtUCopyOut(NativeRipPacket *net_packet,
                   UNetRipPacket *native_packet,
                   int i)
{
        net_packet->rip_rts[i] = native_packet->rip_rts[i];
}



/* return size of a single route in network format */
int ripSizeNetRoute()
{
   return sizeof(NetRipRoute);
}
/* return size of a network packet: note sizeof will return a value
   equal to the size of a rip packet with ONE route */
int ripSizeNetPacket()
{
   return sizeof(NetRipPacket);
}

%%

/*
 * because USC does not support variable length arrays, for loops or
 * check to see if a pointer is more aligned than its annotation
 * states the following C code is needed
 */

#define  ripSizeNetPreamble() (ripSizeNetPacket() - ripSizeNetRoute())

/*
 * Copy a RIP packet from network format to native format
 * and return the number of routes in the packet
 */
```

```
int ripCopyPktIn(void *native_ptr, void *net_ptr, int pkt_size)
{
    int num_routes;
    int i;

    num_routes = (pkt_size-ripSizeNetPreamble())/ripSizeNetRoute();

    /* with high probability rip packet is word aligned in the buffer */
    if (( ((int)net_ptr) % 4) == 0) {
        ripFixCopyIn(native_ptr, net_ptr);
        for (i=0; i<num_routes; i++)
            ripRtCopyIn(native_ptr, net_ptr,i);
    } else {
        ripFixUCopyIn(native_ptr, net_ptr);
        for (i=0; i<num_routes; i++)
            ripRtUCopyIn(native_ptr, net_ptr,i);
    }
    return num_routes;
}

/*
 * Copy a RIP packet from native format to network format
 */

void ripCopyPktOut(void *native_ptr, void *net_ptr, int num_routes)
{
    int i;

    /* with high probability rip packet is word aligned in the buffer */
    if (( ((int)net_ptr) % 4) == 0) {
        ripFixCopyOut(native_ptr, net_ptr);
        for (i=0; i<num_routes; i++)
            ripRtCopyOut(native_ptr, net_ptr,i);
    } else {
        ripFixUCopyOut(native_ptr, net_ptr);
        for (i=0; i<num_routes; i++)
            ripRtUCopyOut(native_ptr, net_ptr,i);
    }
}
/*
 * given the number of routes in the packet return the size in bytes
 * of the buffer needed to hold the packet in network form
 */
int ripPktSize(int num_routes)
{
    return  ripSizeNetPreamble() + (num_routes * ripSizeNetRoute());
```

```
}
```

This program includes the constants needed to define the RIP packet and the USIT generated annotated data types. Thus to make this work rip.usc needs to be run through CPP before invoking USC on the code.

The file rip.usc contains both USC stubs and C function definitions. This is necessary because USC does not support for-loops in its stub syntax and a for-loop is needed to do variable length array copies. Therefore the for-loop is moved into a C function which then calls a USC stub for each iteration. The C stub also does a simple optimization (checking for better than annotated alignment) which is not currently done by the USC code generator. We do this optimization because we know with high probability that the buffer the message is written in will be four byte aligned.

6 Differences with USC 1.0

This section is intended to help people who know how to use USC 1.0. Please note that the USC 2.0 is not backward compatible with the previous version.

6.1 USC

While USC 2.0 is was created primarily to improve the quality of the code base, we took the opportunity to fix some of the more glaring problems with the syntax. The major changes to USC include:

- The USC usage and command line options have changed.

- Target machine characteristics are specified using a form of typedef, replacing the #pragma used in USC 1.0.

-Annotations can now omit the "Msize" value when it matches the "Tsize" value, as is usually the case. In the new notation, if the msize differs from the tsize, it is given using the following syntax:

```
(Tsize/Msize, alignment, <order>).
```

- To define a bit field only a bit order annotation is needed after the bit size. For example:

```
int x2 (2, 2, <0..1>) : 10 <4..15>;
```

-Integral constants and simple expressions may be assigned to struct members.

-Variable length arrays are no longer supported.

-USC and USIT have been modified to accept CPP input.

6.2 USIT

USIT has undergone major changes. In the previous version USIT took a USC program and produced a USC program. These programs differ in the fact that any un-annotated type definitions are annotated with the USC annotations corresponding to the native layout used by a particular compiler/machine combination.

USIT 2.0 takes a file containing list of un-annotated C typedefs (in C syntax) and produces a file which contains those typedefs annotated with the USC annotations corresponding to the native layout used by a particular compiler/machine combination. This output is not a valid USC program but it can be included into a USC program using the include syntax defined above. Note that now the input to USIT is a perfectly legal C include file.

This change was made to simplify USIT and reduce the number of typedefs that would have to be changed when making a change to a structure. Before, if you changed the name of a field in a structure that USC generated stubs for, you would have to make the change in three places: the C include file where that type was defined, the USC file where the native layout was defined, and the USC file where the non-native layout you are converting to was defined. With the new syntax, the need to change two typedefs in the USC file is eliminated.

7 Reference

Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. USC: A Universal Stub Compiler. *Proceedings of SIGCOMM 94 Symposium,* October, 1994.