The USC Manual[1]

Sean O'Malley, Todd Proebsting, Allen Brady Montz, and Dorgival Guedes
University of Arizona[2]

TR 94 26

Abstract

USC is a new 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. This document is the USC reference manual.

October 11, 1994

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' address: Department of Computer Sciences, University of Arizona, Tucson, AZ 85721. Email: {sean,todd,bradym,dorgival}@cs.arizona.edu.

1 Introduction

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 */
#pragma long(4,4,4,<0..3>)
#pragma int(4,4,4,<0..3>)
#pragma short(2,2,2,<0..1>)
#pragma char(1,1,1,<0>)

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

/* tcp header in network format */
typedef struct tcp_net_hdr {
  unsigned short   sport(2,2,0,<1..0>),
           dport(2,2,2,<1..0>);
  unsigned int     x2(1,1,4,<0>):4(4,8,0,<4..7>),
           off(1,1,4,<0>):4(4,8,0,<0..3>);
  unsigned long    seq(4,4,5,<3..0>),
           ack(4,4,9,<3..0>);
  unsigned char    flags(1,1,13,<0>);
  unsigned short   win(2,2,14,<1..0>),
           sum(2,2,16,<1..0>),
           urp(2,2,18,<1..0>);
} net_hdr(0,20,1,0);

/* a stub definition */
void tcphdr(net_hdr *src, native_hdr *dest)
{
  *dest = *src;
}
```

```
%%
```

A USC program consists of a series of pragmas followed by type and stub definitions. The pragmas 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 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.

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 functions implementations of of all remaining USC stubs. As an option (-h 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 (-p) the user may have USC generate functions whose arguments types are identical to those found in the USC stub definitions. To be compilied 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.

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 (enumerated types are NOT supported in this release). 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 currently 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.

While C does not support the declaration of variable length arrays, C programmers often get around this restriction by allocating arrays larger than that defined in the type. To support this, the USC type system is modified to allow the user to declare a variable length array. Variable length arrays may appear standalone or as the last element of a structure. A variable length array is defined as follows:

```
int a[name];
```

where name is a C variable name. The name used in the type definition must correspond to an integer parameter in the USC stub definition. This parameter is used to pass the actual size of the array to the stub. The sizeof a type containing a variable length array is defined as the size of that type with an array of length 1.

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 defined type. The input file must precisely specify the correct byte order and offset of every type. Pragmas are used to inform USC of the native format in the compiler/host combination that will be used to compile and execute the generated stub.

All USC annotations are lists of four properties. The exact properties in the list is determined by context. A USC annotation found after a variable or parameter name is defined as follows:

type var(tsize, msize, alignment/offset, byte order);

Where tsize is the number of bytes needed to represent the data type and msize the number of bytes the compiler has allocated to store this data type. Tsize must be less than or equal to msize. The third field is interpreted as an alignment in all cases except that of a field in a structure of union, in which case it is interpreted as the offset of that field within the immediately enclosing structure or union. The alignment field is a guarantee to the compiler that the address of the annotated variable modulo alignment is 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 byte order field is used to specify which memory bytes, in what order, are used to represent a given type. The syntax of the byte order field is 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. The USC type annotation for a 4 byte word aligned big endian integer is:

```
int a(4,4,4,<3..0>);
```

When describing a structure, the byte order field must be zero. USC derives the actual information from the annotations of the structure's fields. This notation can describe any C array type. For example 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];
```

The annotations found after a field name are identical to the annotations found after a variable except that the third element specifies the exact offset in bytes from the beginning of the structure. Given the offset USC can determine the alignment of any field in a structure from the alignment of the structure.

USC uses two separate annotations to describe bit-fields. Each bit-field name is annotated with the format of the underlying integer type. If several bit-fields are contained in the same integer they will have the same offset. After the bit-field size specifier another annotation specifies which bits in the underlying integer make up the bit-field. This annotation is analogous to the previous one, except that all of the values are in bits rather than bytes and the offset field must be zero. Note that the bit order of a bit-field is described relative to the byte order of the underlying integer type. For example two four bit bit-fields arranged in the same byte in little endian bit order at offset 4 from the beginning of a structure would be defined as follows:

```
 u_int  x2  (1,1,4,<0>):4(4,8,0,<4..7>),
        off (1,1,4,<0>):4(4,8,0,<0..3>);
```

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.

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 redefine the alignment of the tcp network header to four byte aligned.

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

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. If a field in the type annotation is zero USC will not change the existing value of that field.

3.4 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 and their annotations differ only in byte ordering. Two USC types are identical if they are type compatible and have identical annotations.

3.5 Pragma Declarations:

```
#pragma long (4,4,4,<0..3>)
```

The pragma above defines the layout of a long on the compiler/machine combination for which USC will generate code. USC will take this definition to mean that longs are 4 bytes long, require 4 bytes of storage, and are aligned on 4 byte boundaries. The last field, which could also have been written $\langle 0, 1, 2, 3 \rangle$, means that the least significant byte of a long is at offset 0 from its address, and the most significant byte is at offset 3 from its address. In other words, longs are little endian. Note that the current version of USC assumes that the maximum register size of a given machine is equal to the size of the largest integer type given in a pragma definition.

3.6 USC Stub definitions

Stubs are defined in USC as functions are defined in ANSI C. 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 USC stubs are defined using a restricted subset of the ANSI C statement grammar. Only expression statements 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 to structures with different layouts.

In the USC expression grammar component selection (`->` and `.`), array subscription(`[ ]`), indirection (`*`), sizeof and address of (`&`) are supported on all appropriate types. The assignment operation is supported between all type compatible USC types. Unlike C, assignment between array types is supported. 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 pragmas at the beginning of the USC program. Note that expressions and constants are only allowed in array subscripts. Thus you cannot use USC to generate a stub which initializes a USC structure with constant values.

Parameters to USC stubs must be either pointers to any USC type, or a type 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. USC requires stub parameters that give the lengths of any variable length array arguments. The code below copies one variable length character array to another. len gives the current length of the arrays. While unannotated base types used as the arguments or return value or USC stubs are assumed to be native; we strongly recommend that every type used in parameter list or return value be defined by a USC typedef.

```
typedef int(4,4,4,<0,1,2,3>) NativeInt;
typedef char(1,4,4,<0>) Array[len];


void foo (NativeInt len, Array str1, Array str2)
{
  str1 = str2;
}
```

Any USC stub that contains a variable length array definition as a parameter must have a native integer parameter whose name matches the name given in the array definition.

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. Currently the qualifier macro may only be used on stubs returning void.

The stub tcphdr defined in Figure 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. A stub that peeks into a TCP header in network format and returns the offset field in a four byte, big endian integer would be defined as follows:

```
int(4,4,4,<3,2,1,0>) tcpgetoff (net_hdr *hdr)
{
  return hdr->off;
}
```

3.7 In-Place Data Modification

Currently USC does not support the inplace modification of data types.

3.8 Failure Model

Generated stubs that raise runtime exceptions are very unpopular. USC generated stubs will not raise an exception if the following conditions are met: the alignment and offset information given in the USC source are correct, the user uses the USC sizeof operator to compute buffer length, and all values passed to USC stubs match their definitions in type and format. Because USC does not support pointer types this failure model requires no error checking in the stubs, which improves performance.

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 C type and variable declarations without any USC annotations and outputs a USC program with those types and variables properly annotated for the local compiler/host pair. USIT generates and runs a C program to infer the annotations.

USIT's input format is identical to that of USC except that types may be left un-annotated and no pragma definitions are necessary. USIT will scan its input looking for annotated types and generates a new file with those types annotated with the local host/compiler representation. If no pragma definitions are present USC will generate pragma's to represent the base types of the give host/compiler.

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 execute 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
```

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/USIT. The file also contains the interface between the USC generated stubs and the x-kernel message tool.

The file lets USIT define the formats of the native types. The file will be run through usit on the machine for which the given x-kernel is being built and then run through USC to generate the final C file. USIT was also used to generate the annotations for the network form. Because on the SPARC the network form of the RIP packet is the same as the native format, one can simply run USIT on the RIP packet definition on the SPARC and copy the output into this file.

Since the RIP packet contains a variable length array of routes the variable length support in USC was used. Because of the limitations of USC syntax the conversion of packet length into number of routes in the packet is somewhat awkward (and probably inefficient.).

Since the x-kernel message tool insures that the RIP header will be four byte aligned except for certain rare cases this program defines two different sets of input and output stubs: one for four byte aligned network headers and one for unaligned network headers.

```
/* standard C code */

/* the standard x-kernel include files */
#include "xkernel.h"
#include "ip.h"
#include "vnet.h"
#include "arp.h"
#include "udp.h"
#include "riproute.h"
#include "riproute_i.h"
#include "rip.h"
#include "rip_i.h"
extern int traceripp;
%%
/* USC/USIT code */

/* USIT will generate the necessary pragma's for each machine */

/* USIT will annotate the following types */
/* an integer: host format */
typedef int NativeInt;
/* an IP address: host format */
typedef struct iphost {
    unsigned char      a,b,c,d;
} NativeIpHost;
/* a RIP route: host format */
typedef struct  riprt {
        short   rr_family;      /* 4BSD Address Family                */
        short   rr_mbz;         /* must be zero                       */
        NativeIpHost  rr_addr;  /* ip address                         */
        char    rr_mbz2[8];     /* must be zero */
        int     rr_metric;      /* distance (hop count) metric          */
} NativeRipRoute;
/* RIP packet structure: host format */
typedef struct  rip {
        char    rip_cmd;        /* RIP command                          */
        char    rip_vers;       /* RIP_VERSION, above                   */
        short   rip_mbz;        /* must be zero                         */
        NativeRipRoute   rip_rts[length];
} NativeRipPacket;

/* the format of the following types is defined by RFC-1058 */
/* RIP packet structure: network format */
typedef struct {
  signed char rip_cmd(1, 1, 0, <0>);
  signed char rip_vers(1, 1, 1, <0>);
  signed short rip_mbz(2, 2, 2, <1,0>);
  struct {
    signed short rr_family(2, 2, 0, <1,0>);
    signed short rr_mbz(2, 2, 2, <1,0>);
    struct {
      unsigned char a(1, 1, 0, <0>);
      unsigned char b(1, 1, 1, <0>);
      unsigned char c(1, 1, 2, <0>);
      unsigned char d(1, 1, 3, <0>);
    } rr_addr(0, 4, 4, 0);
    signed char rr_mbz2(1, 1, 8, <0>)[8];
    signed int rr_metric(4, 4, 16, <3..0>);
  } rip_rts(20, 20, 4, 0)[length];
} NetRipPacket(0, 24, 4, 0);
/* redefine the network rip packet to be one byte aligned */
typedef NetRipPacket UNetRipPacket(0,0,1,0);

/* copy the rip packet into native form */
void ripCopyIn(NativeRipPacket *native_packet,
               NetRipPacket *network_packet, int length)
{
   *native_packet = *network_packet;
}
/* copy the rip packet into network form */
void ripCopyOut(NetRipPacket *network_packet,
                NativeRipPacket *native_packet, int length)
{
   *network_packet = *native_packet;
}
/* the same as above only assume network packet is unaligned */
```

```
void ripCopyUIn(NativeRipPacket *native_packet,
                UNetRipPacket *network_packet, int length)
{
   *native_packet = *network_packet;
}
void ripCopyUOut(UNetRipPacket *network_packet,
                 NativeRipPacket *native_packet, int length)
{
   *network_packet = *native_packet;
}
/* return size of a single route */
NativeInt ripSizeofNetRoute()
{
   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 */
NativeInt ripSizeofNetPacket()
{
   return sizeof(NetRipPacket);
}
%%
/* standard C code */

/*
 * This works because the sizeof a structure containing a variable
 * length array is equal to the size of the same structure with an
 * array of length 1
 */
#define  ripSizeofNetPreamble() (ripSizeofNetPacket() - ripSizeofNetRoute())
/*
 * ripHdrStore -- write header to potentially unaligned msg buffer.
 */
void
ripHdrStore(hdr, dst, len, arg)
    VOID *hdr;
    char *dst;
    long int len;
    VOID *arg;
{
    int num_routes;
    num_routes = (len-ripSizeofNetPreamble())/ripSizeofNetRoute();
    xTrace1(ripp, TR_EVENTS, "ripHdrStore entered num_routes=%d",num_routes);

    /* with high probability dst will be aligned but you never know */
    if (( ((int)dst) % 4) == 0) {
        ripCopyOut(dst,(char *)hdr, num_routes);
    } else {
        ripCopyUOut(dst,(char *)hdr, num_routes);
    }
}
/*
 * ripHdrLoad -- load header from potentially unaligned msg buffer.
 */
long
ripHdrLoad(hdr, src, len, arg)
    VOID *hdr;
    char *src;
    long int len;
    VOID *arg;
{
    int num_routes;
    xTrace0(ripp, TR_EVENTS, "ripHdrLoad entered");
    if ((len-ripSizeofNetPreamble()) % ripSizeofNetRoute() != 0) {
        xTrace0(ripp, TR_ERRORS, "ripHdrLoad not right size");
        return 0;
    }
    num_routes = (len-ripSizeofNetPreamble())/ripSizeofNetRoute();
    xTrace1(ripp, TR_EVENTS, "ripHdrLoad num_routes=%d",num_routes);
    if (num_routes < 1) {
        xTrace0(ripp, TR_ERRORS, "ripHdrLoad no routes");
        return 0;
    }

    /* with high probability src will be aligned but you never know */
    if (( ((int)src) % 4) == 0) {
        ripCopyIn((char *)hdr, src, num_routes);
    } else {
        ripCopyUIn((char *)hdr, src, num_routes);
    }
    return len;
}
```

6 Known Problems

The inability of USC to define C macro's which return values limits the usefulness of the sizeof and address of operations. In addition through an oversight the current USC syntax does not allow the use of constants anywhere but inside of an array subscrypt. This implies that it is impossible to use USC to initialize a structure with constant values. The following is not valid USC syntax:

```
/* initialize a tcp header */
void tcphdrinit(net_hdr *hdr)
```

```
{
    hdr->urp = 0;
    hdr->flags = 0;
    hdr->wnd = 4*1024;
}
```

The USC code generator generates poor code for types whose alignment is less than the register size on the machine. This is especially noticeable on the DEC Alpha. We are currently modifying the USC code generator to fix this. USC's sign extension code is also less than optimal and again we are working on a fix.

Currently USIT cannot be used to define mixed types. Thus you can't use USIT to define a ip native header in which the the IP host addresses are kept in network byte order to save processing time.