Waste Not, Want Not— A Proposal for Shorter IORs

Michi Henning, Michael Neville
{michi,neville}@dstc.edu.au
CRC for Distributed Systems Technology (DSTC)
University of Queensland, Qld 4072, Australia

Abstract

Many CORBA services (such as the Naming Service and Trading Service) need to store large numbers of Interoperable Object References (IORs) as strings in a database. CORBA 2.1 defines an interoperable string representation for IORs which results in strings that are needlessly large. This paper describes an alternative encoding for stringified IORs which is more compact, retains the printable nature of stringified IORs, is extensible to allow other encodings to be added in the future, and can co-exist with the current encoding in the same ORB. We also provide an implementation of the proposed encoding.

1 INTRODUCTION

The CORBA 2.1 specification [2] defines a string representation for *Interoperable Object References* (*IORs*). Such *stringified* IORs consist of the prefix **`IOR:`** followed by an even number of hexadecimal digits. The digit sequence following the prefix is created by treating the *Common Data Representation* (*CDR*) of an IOR as a sequence of octets, and encoding each octet as a pair of hexadecimal digits for the high-order and low-order nibble. A typical IOR looks like:

```
IOR:000000000000002049444C3A68702E636F6D2F48504F52425F4F626A4C
6F6361746F723A312E300000000002000000010000000A4000000000000000007
0000000A00000002300000000810400000000008534F413A312E3000000000
0B000000274F4C34656538366165302D636665302D373164302D316532382D
38323636623033353030303000000000000C00000001010000000810400200
0000000810400300000016626F626F2E647374632E6564752E61753B313539
3700000008104803000000143133302E3130322E3137362E35333B31353937
00000000000000005400010000000000011626F626F2E647374632E6564752E
6175000000063D000003448503A534F413A312E300030004F4C346565383661
65302D636665302D373164302D316532382D38323636623033353030303000
```

The exact length of a stringified IOR varies from ORB to ORB and from object to object—it depends on the number of protocols supported by the target object and the size of the object key embedded in the reference. Lengths of 300–700 bytes are common.

CORBA defines a distinguished *nil* IOR (which denotes no object). The nil IOR is the same for all ORBs, but has two representations for big-endian and little-endian format:

`IOR:00000000000000010000000000000000` (big-endian)

`IOR:01000000010000000000000000000000` (little-endian)

The current representation of stringified IORs is wasteful. Because each CDR octet maps to two bytes in the resultant string, 4 bits in every byte are unused, so the actual information content of a stringified IOR is at most 50%. Further, IORs contain an uneven distribution of hexadecimal digits (for example, there are usually long sequences of zeros), and a typical stringified IOR contains around 65% redundant bits.

Frequently, servers need to store large numbers of IORs in a database for later retrieval. The CORBA Naming Service and Trading Service are examples of such servers—in essence, they are repositories of IORs where each IOR can be retrieved according to some description. For scalable implementations that need to store large number of IORs, the size of each stringified reference becomes significant. For example, stringified IORs in a Trading Service can account for around 60% of the total database size. The excessive size of stringified IORs not only results in wasted disk space, but also limits performance because of increased I/O activity to the database.

The following sections describe an alternative format for representing IORs as strings. The suggested format:

- reduces the size of stringified IORs by around 45%,
- results in well-behaved printable strings,
- generates IORs that are self-identifying,
- allows both the current format and the proposed format to co-exist in an ORB,
- is extensible to permit future addition of other encodings.

2 THE PROPOSED ENCODING

2.1 Basic Encoding

The format suggested here uses a base-64 encoding instead of base-16. This means that each byte of a stringified IOR contains 6 bits of information of the original CDR, instead of just 4 bits. Base-64 encoding therefore reduces the size of a stringified IOR by 33%.

The encoding maps input CDR 6-bit blocks onto target characters as follows:

CDR input value	Encoded char
0	`0`
1	`1`
...	...
9	`9`
10	`a`
11	`b`
...	...
35	`z`
36	`A`
37	`B`
...	...
61	`Z`
62	`-`
63	`+`

This particular base-64 encoding was chosen to ensure that stringified IORs are well-behaved. In particular, the encoding guarantees that stringified IORs do not contain white space, control characters, or newline characters. Maintaining this guarantee is important because application code expects stringified IORs to be suitable for line-oriented I/O, text I/O, or tokenizing. The chosen characters are unlikely to be interpreted as meta-characters by command line interpreters and permit a reference to be represented as an ASN.1 printable string [1].

With this mapping, each three-byte sequence in CDR maps to a four-byte sequence in the stringified IOR (whereas the standard encoding generates a six-byte sequence for every three bytes of CDR). An example is shown in Figure 1:

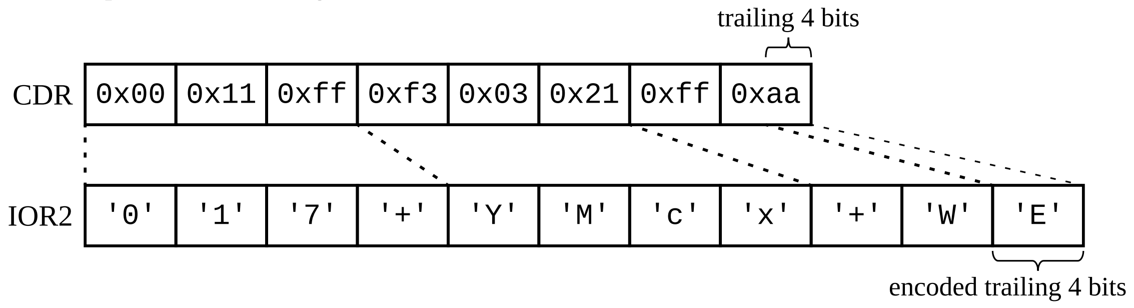

Figure 1 Example encoding from CDR to new format

The encoding repeatedly peels off six consecutive bits of the CDR representation and generates the corresponding byte in the new encoding using the above mapping. Depending on the length of the CDR representation, the encoding may terminate on a byte boundary, or find two or four trailing bits that still need encoding. These boundary conditions are handled as follows:

- termination on a CDR byte boundary:

  The final byte is encoded using the normal base-64 encoding.

- four trailing CDR bits:

  The remaining four bits are padded with two zero bits on the right. The encoded byte is then generated using the padded value in the normal base-64 encoding.

- two trailing CDR bits:

  The remaining two bits are padded with four zero bits on the right. The encoded byte is then generated using the resulting value for the normal base-64 encoding.

The encoding of trailing bits in this way requires attention when decoding a stringified reference—the decoder needs to keep track of how many six-bit blocks have been decoded so far and needs to treat the final byte of the stringified IOR as a special case.

2.2 Encoding Runs of Zeros

The CDR of an object reference often contains long runs of zeros. The stringified reference shown on page 1 contains several such runs (each zero digit in the IOR corresponds to four consecutive zero bits in the CDR). The proposed new format uses a run-length encoding to reduce the size of the generated stringified IOR by modifying the basic encoding shown above.

If the CDR contains three or more consecutive 6-bit blocks where all bits are zero, the stringified format contains an ‘`=`’ character followed by a single-byte count. The ‘`=`’ serves to mark the start of a run (it is not used for the basic base-64 encoding). The byte that follows the ‘`=`’ encodes the number of 6-bit blocks of zeros. The byte value is determined by subtracting 3 from the number consecutive zero blocks and writing the resulting number in base-64 encoding. This generates the following mapping:

Run length	Char sequence
3	`=0`
4	`=1`
5	`=2`
...	...
12	`=9`
13	`=a`
14	`=b`
...	...
38	`=z`
39	`=A`
40	`=B`
...	...
64	`=Z`
65	`=-`
66	`=+`

Encoding runs of zeros in this way permits up to 66 blocks to be encoded in two bytes of the stringified IOR. If runs of zeros are longer than 66 blocks, the stringified IOR contains several adjacent encoded zero runs. For example, a run of 88 zero blocks in CDR results in the character sequence **`=+=9`** in the stringified IOR.

2.3 Identification

The new encoding uses the prefix **`IOR2:`** to distinguish new references from old ones. Further, prefixes are reserved to allow other encodings to be added in the future:

Prefix	Use	Encoding number
`IOR:`	current encoding	1
`IOR2:`	proposed encoding	2
`IOR3:`	reserved	3
...	...	...
`IOR9:`	reserved	9
`IOR10:`	reserved	10
...	...	...
`IOR99:`	reserved	99
`IOR100:`	reserved	100
...	...	...

This identification scheme allows IORs in different encodings to be distinguished from each other. The CORBA 2.1 specification uses the prefix **`IOR:`**. The new encoding proposed here uses the prefix **`IOR2:`**. To allow other encodings to be added in the future, prefixes of the form **`IOR[0-9]+:`** are reserved.

2.4 Recording Original CDR Size

The IOR2 encoding uses a four-byte count following the **`IOR2:`** prefix. This count (in base-64 encoding) records the number of bytes in the original CDR. As an example, a CDR of 16 bytes results in a count value of **`000g`**, and a CDR of 308 bytes results in a count value of **`004Q`** (4 * 64 + 52). With this four-byte count, the IOR2 format can encode IORs with a CDR representation of up to 16 MB in length.

The count is useful because it allows the caller to determine the number of bytes required for the CDR representation of an IOR2 as a constant-time operation. This avoids the need to work with fixed-size buffers or to make an additional scan over an IOR2 to determine its CDR size.

2.5 Overall Format

The overall IOR2 format is shown in Figure 2:

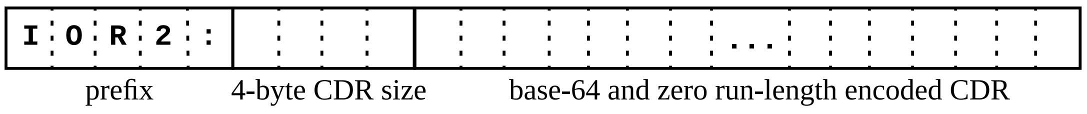

Figure 2 IOR2 format

In IOR2 representation, the IOR shown on page 1 becomes:

```
IOR2:004Q=6219h4MWq70KoSZJbQxgjR92nQZyqANLoS5QrT8WciUM=38=11=1
F=77=12w=18M=1211=4xjjQ4WciUM=3I=1DjQMQpmkUdC5Bc2RzpCkMbjsNp30
JcmkOe2QUczoSoz0Pdj0Mc3=5M=110g=1wgg08=221100M=01pyrS9LbChPt6c
KpmhRbC5ReP4Rejs=1844w3=1534Pc2UNc38KcjsSbzkPeP4Rejs=75g00g=34
m9LoCYKp7dQoOVBp7kKonk=0oZ=1d4xgeBdfgjENbz00c01fj3hBpjwSomkMbm
dCpj0JdP5Ac2QNpj8UbjwOdzpyc3cRc30Mc00
```

This IOR is 347 bytes long (compared to 620 bytes in the standard encoding), which corresponds to a compression ratio of 44%.

A big-endian nil reference in IOR2 format is represented as the 14-byte string

```
IOR2:000g=74=8
```

This represents a saving of 61% compared to the standard encoding.

A little-endian nil reference in IOR2 format requires 16 bytes, giving a saving of 56%:

IOR2:000g0g=14=c

3 CONVERSION EFFICIENCY

The example implementation shown in Section 8 achieves around 10000 conversions per second for IORs with a CDR length of 212 bytes. Conversion speed decreases linearly with the size of the IOR. The example implementation sacrifices performance to remain portable and to keep the object code small. If portability or code size is not an issue, performance can be improved substantially. Even so, at 10000 conversions per second, this implementation does not noticeably impact on ORB performance.

4 COMPATIBILITY ISSUES

An ORB modified for IOR2 encoding can successfully decode both the current and the IOR2 format because of the different prefixes. This means no compatibility issues arise for modified ORBs (they can simply support both formats).

However, the opposite is not true. If an IOR2 reference is passed to code using a CORBA 2.1 compliant ORB, decoding will fail.

Stringified IORs are typically used for two purposes:

- Configuration

  Stringified references are often used to configure start-up services (such as the Naming Service) by writing the reference into a configuration file or repository. It is easy to ensure at this point that a reference of the correct format is used, so the incompatibility presents no problems.

- Persistence

  The main use of stringified object references appears in applications such as the Naming Service or the Trading Service, where the server needs to store an object reference persistently on secondary storage for later retrieval. The only way to achieve this portably is to stringify the reference. However, the actual string format does not matter in this case, since the string is treated as an opaque value by the server, so no compatibility issues arise.

Apart from these two uses, stringified IORs are almost never exchanged. Instead, IORs are passed as parameters and return values of IDL operations (where no conversion to string format ever takes place). In the rare case where IOR2 references need to be passed as strings to applications which cannot decode them, the IOR2 references can easily be converted back to the standard format (the example implementation in Section 8 illustrates this).

For these reasons, we do not consider incompatibility to be a practical problem.

5 CONCLUSIONS

Our testing with object references for different ORBs shows that the proposed IOR2 encoding reduces the size of stringified references by around 45%. The encoding leaves approximately 20% redundancy in a stringified IOR. However, compressing IORs to reclaim those last 20% of redundancy would require a binary representation and destroy the printable nature of stringified IORs. The IOR2 encoding is a compromise which removes most of the redundancy, yet retains a simple text representation.

The savings in size are worthwhile. During experiments with DSTC's Trading Service [3], we found that a 100 MB database shrank to 73 MB when we stored IOR2 encoded references. This not only reduced the amount of disk space required, but resulted in better performance due to reduced I/O bandwidth (many trader operations are directly I/O bound, so a 25% reduction in I/O results in almost 25% performance improvement).

The IOR2 encoding can be added to a future revision of CORBA and co-exist with the current encoding. If adopted as a standard, the benefits of reduced IOR size will be available to CORBA developers automatically.

Without standardization, IOR2 encoded strings cannot be exchanged portably between applications. However, the IOR2 encoding is still useful as an implementation technique for servers that need to store and retrieve large numbers of object references as strings of text.

As evidenced by our example implementation, the performance of encoders and decoders is not an issue. Even though our implementation is not optimized so it can remain generic, its performance does not appreciably affect conversion between CDR and string representations.

6 ACKNOWLEDGEMENTS

Thanks go to Martin Chilvers, Rich Salz, and Steve Vinoski, who commented on an earlier version of this paper and suggested improvements. The work reported in this paper has been funded in part by the Cooperative Research Centre Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

7 REFERENCES

[1] CCITT Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1), 1988.

[2] The Common Object Request Broker: Architecture and Specification, Object Management Group, Revision 2.1, August 1997.

[3] DSTC Trading Service—An OMG conformant Trading Service Implementation, http://www.dstc.edu.au/AU/projects/corba-trader/fact_sheet.html.

8 EXAMPLE IMPLEMENTATION

The example implementation consists of the actual encoding and decoding functions, as well as a simple test program which converts current-style IORs into IOR2 format and vice-versa:

```
size_t cdr_to_ior2(
        const void *cdr,
        size_t cdrsize,
        char *ior2,
        size_t ior2size);
```

This function converts the CDR in `cdr`, of length `cdrsize` to IOR2 format. The resulting string is written to `ior2`. At most `ior2size` bytes (including the terminating NUL byte) are written to `ior2`. The stringified IOR consumes at most 4⁄3 `cdrsize` + 10 bytes.

If the conversion is successful, the return value is the number of bytes written to `ior2` (including the terminating NUL byte). If `ior2` is too short to hold the stringified IOR, the return value is zero.

```
size_t ior2_to_cdr(
        const char *ior2,
        void *cdr,
        size_t cdrsize);
```

This function converts the stringified IOR in `ior2` into CDR. The result is written to `cdr`. At most `cdrsize` bytes are written to `cdr`. The `cdr_size()` function can be used to determine the number of bytes required for decoding in advance as a constant-time operation.

If the conversion is successful, the return value is the number of bytes written to `cdr`.

Conversion fails and the return value is zero if

- `ior2` does not have an **`IOR2:`** prefix
- the length field of the IOR is zero
- `cdr` is too short to hold the result

```
size_t cdr_size(const char *ior);
```

This function returns the number of bytes of CDR required to decode the reference passed as `ior`. Both current and IOR2 encoded references can be passed.

The return value is zero if `ior` does not start with a valid prefix (**`IOR:`** or **`IOR2:`**).

8.1 ior2.h

```
#ifndef IOR2_H
#define IOR2_H

#include <stddef.h>
```

```
extern size_t   cdr_to_ior2(const void *, size_t, char *, size_t);
extern size_t   ior2_to_cdr(const char *, void *, size_t);
extern size_t   cdr_size(const char *);

#endif /* IOR2_H */
```

8.2 ior2.c

```
/*
 * Copyright (C) DSTC Pty Ltd (ACN 052 372 577) 1997.
 *
 * Any party obtaining a copy of this file from DSTC Pty Ltd, directly
 * or indirectly, is granted, free of charge, a full and unrestricted
 * irrevocable, world-wide, paid up, royalty-free, nonexclusive right and
 * license to deal in this software (the "Software"), including without
 * limitation the rights to use, copy, modify, merge, publish, distribute,
 * sublicense, and/or sell copies of the Software, and to permit persons
 * who receive copies from any such party to do so, with the only
 * requirement being that all copyright notices remain intact.
 *
 * This software is being provided "AS IS" without warranty of any kind,
 * express, implied or otherwise, including without limitation, any
 * warranty of merchantability or fitness for a particular purpose.
 *
 * In no event shall DSTC Pty Ltd be liable for any special, incidental,
 * indirect or consequential damages of any kind, or any damages whatsoever
 * resulting from loss of use, data or profits, whether or not advised of
 * the possibility of damage, and on any theory of liability, arising out
 * of or in connection with the use or performance of this software.
 *
 * DSTC Pty Ltd welcomes comments and bug fixes related to this software.
 * Please address any queries, comments or bug fixes (please include
 * the name and version of the software you are using and details
 * of your operating system version) to the address below:
 *
 *       DSTC Pty Ltd
 *       Level 7, Gehrmann Labs
 *       University of Queensland
 *       St Lucia, 4072
 *       Tel: +61 7 365 4310
 *       Fax: +61 7 365 4311
 *       Email: Enquiries@dstc.edu.au
 */

/*-------------------------------------------------------------------------*/

/*
 * Revision history:
 *
 * 15 Aug 97: Changed init_codevals() to use a loop for initialization
 *            instead of in-line code. Thanks to Rich Salz for suggesting
 *            this.
 *
 * 20 Oct 97: Changed prefix to add a colon separator. Thanks to Steve
 *            Vinoski for pointing out that this was necessary for
 *            use of IORs in web browsers.
 *
 *            Added example for little-endian nil IOR. Thanks to Martin
 *            Chilvers for pointing out that there are two representations
 *            of nil IORs.
 */
```

```
/*---------------------------------------------------------------------------*/

/*
 * cdr_to_ior2(),
 * ior2_to_cdr()        - convert from CDR to IOR2 format and vice versa
 *
 * cdr_size()           - return the CDR size of an IOR (both old and new)
 *
 * IOR2 encoded strings are typically 40% - 45% shorter than the old IOR
 * format, depending on the specific IOR (compression improves for
 * longer IORs).
 *
 * Big-endian nil IORs are compressed by 61%, from
 * "IOR:00000000000000010000000000000000" to "IOR2:000g=74=8".
 *
 * Little-endian nil IORs are compressed by 56%, from
 * "IOR:01000000010000000000000000000000" to "IOR2:000g0g=14=c".
 *
 * An IOR in IOR2 format is a well-behaved string - it contains only digits,
 * lower and upper case letters, and the letters ':', '-', '+' and '='.
 * This means it can easily be used on the command line, as a database
 * field value, or for line-oriented file I/O (the stringified IOR does
 * not contain embedded control characters, white space, or newline characters).
 * IOR2 references can also be represented as an ASN.1 printable string.
 *
 * It is possible to compress IORs further, by using a Lempel-Ziv or
 * similar encoding, but this sacrifices the printable nature of the string
 * (binary values need to be used to get the last few percent of compression).
 * The algorithm used here is a compromise - it leaves around 20% redundancy
 * in the compressed IOR, but keeps the resultant string printable.
 *
 * The code below makes no assumptions about codesets, so it works for both
 * ASCII and EBCDIC. There are also no assumptions about character
 * ordering (or groups of characters occupying adjacent code positions),
 * so this source can be used with any single-byte code set.
 * To remain portable, the code table is initialized at run-time.
 *
 * The code compiles cleanly with both ANSI-C and C++ compilers.
 * It works with both signed and unsigned char implementations and
 * makes no assumptions about byte ordering. 2's complement representation
 * is *not* assumed. The code will work on 16 bit machines and is suitable
 * for use in free-standing implementations (it does not require libc or
 * other libraries).
 *
 * The code is thread-safe and reentrant, except for the run-time
 * initialization of the decode array. To get thread safety for that,
 * the 'once' guard variable and the call to the initialization function
 * (init_codevals) need to be protected by a mutex.
 *
 * Performance of the conversions between CDR and IOR2 (and back) should not
 * be a concern. The code below sacrifices a lot of efficiency to remain
 * portable. Even so, on an HP715/100 workstation, the code achieves:
 *
 * CDR length | Direction   | Conversions/sec
 * ------------------------------------------
 *    308     | cdr_to_ior2 |     6500
 *    308     | ior2_to_cdr |     8500
 *    212     | cdr_to_ior2 |     9500
 *    212     | ior2_to_cdr |    12500
 *
 * So even for a rather long IOR with 308 bytes of CDR, conversion speed
 * won't be an issue.
 *
 * Not surprisingly, the cost of conversion increases linearly with the
 * size of the reference.
 *
 * The algorithm chosen for this implementation minimizes code size
 * (the PA-RISC object code size is 2.5 kB for code and data).
 */

/*--------------------------------------------------------------------------*/

#include        "ior2.h"

/*--------------------------------------------------------------------------*/

/*
 * decode is an array of code values. It maps a character (the index)
 * to a value. The array is initialized at run-time to make sure
 * the code works with both ASCII and EBCDIC (or other codes).
 *
 * The mapping for the decode array is
 *
 *      '0' -> 0
 *      ...
 *      '9' -> 9
 *      'a' -> 10
 *      ...
 *      'z' -> 35
 *      'A' -> 36
 *      ...
 *      'Z' -> 61
 *      '-' -> 62
 *      '+' -> 63
 *
 * The array is sparse and initialized only for the relevant
 * lookup positions (the unsigned byte value is used directly as an index).
 * Again, this gives codeset independence (and improves speed a little).
 */
static int decode[256];

/*
 * Guard to do lazy initialization of decode array.
 */
static int once = 0;

/*
 * Inverse mapping of decode array. Maps a value to a character.
 */
static const char code[] = "0123456789"
                           "abcdefghijklmnopqrstuvwxyz"
                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                           "-+";

/*
 * ZERO_CODE ('=') is a special marker for run-length encoding of
 * consecutive zeros. It must not occur in the codeval or code arrays.
 *
 * Runs of zeros are encoded as:
 *
 *      Number of zeros         Code
 *      ----------------------------
 *               3              =0
 *               4              =1
 *              ...             ...
 *              12              =9
 *              13              =a
 *              14              =b
 *              ...             ...
 *              38              =z
 *              39              =A
 *              40              =B
 *              ...             ...
 *              64              =Z
 *              65              =-
 *              66              =+
 *
 * In other words, the number of zeros in a run is encoded by the byte
 * following the '=' character.
 *
 * For encoding, that byte value is
 *
 *      code[zeros - 3]
 *
 * For decoding, the run length is reconstructed by using the value
 * of the byte following the '=' marker as an index into the decode
 * array and adding 3:
 *
 *      decode[byte] + 3
 *
 * Runs of more than 66 zeros are encoded by two consecutive run-length
 * encodings. For example:
 *
 *      (70 zeros) -> =+=1
 *
 * After each run, at least three zeros must be left for another run,
 * otherwise the normal encoding is used. For example:
 *
 *      (68 zeros) -> =+00
 */
static const char ZERO_CODE = '=';

/*
 * Some powers of two. The code uses div (/) and mod (%) operations to
 * set and clear bits (to avoid assuming 2's complement representation).
 * Making constants here keeps the code readable. We use the preprocessor
 * (instead of real constants) because real constants can slow the code
 * down by more than a factor of two with some compilers.
 */
#define P2_18   262144          /* 2^18, 256 kB */
#define P2_12   4096            /* 2^12, 4 kB */
#define P2_6    64              /* 2^6,  64 bytes */
#define P2_4    16              /* 2^4,  16 bytes */
#define P2_2    4               /* 2^2,  4 bytes */

/*------------------------------------------------------------------------*/

/*
 * Initialize the decode array with the numeric codes for each character.
 * We do this at run time to make it work for both ASCII and EBCDIC.
 */

static void
```

```
init_codevals()
{
        int i;

        for (i = 0; i < sizeof(code) - 1; i++)
                decode[(unsigned)code[i]] = i;
}

/*--------------------------------------------------------------------------*/

/*
 * Convert 'cdrsize' bytes in the 'cdr' buffer into IOR2 format in 'ior2'.
 * 'ior2size' must be set to the size of the 'ior2' buffer.
 *
 * Return value:
 *
 *      - A non-zero return value indicates the actual number of bytes
 *        written into 'ior2' (the count includes the terminating NUL byte).
 *
 *      - A zero return value indicates that the length of the 'ior2'
 *        buffer as specified by 'ior2size' was insufficient to hold
 *        the encoded IOR.
 */

size_t
cdr_to_ior2(const void *cdr, size_t cdrsize, char *ior2, size_t ior2size)
{
        size_t                  val;            /* Temporary */
        unsigned                pos;            /* State variable */
        unsigned                zeros;          /* Num zeros blocks in run */
        size_t                  ior2len;        /* Length of ior2 */
        const unsigned char     *cdrp;          /* First byte of CDR */
        size_t                  cdrlen;         /* Length of cdr */
        int                     last_iter;      /* True for last byte of CDR */

        /*
         * Need 9 bytes for prefix and length field.
         */
        if (ior2size < 9)
                return 0;

        /*
         * Set prefix.
         */
        *ior2++ = 'I';
        *ior2++ = 'O';
        *ior2++ = 'R';
        *ior2++ = '2';
        *ior2++ = ':';

        /*
         * Encode 4-byte CDR length.
         */
        val = cdrsize;
        *ior2++ = code[val / P2_18];
        val %= P2_18;
        *ior2++ = code[val / P2_12];
        val %= P2_12;
        *ior2++ = code[val / P2_6];
        *ior2++ = code[val % P2_6];
```

```
ior2len = 9;    /* Five bytes for prefix, plus four bytes for length */

/*
 * Repeatedly peel off the next six CDR bits.
 */
cdrp = (const unsigned char *)cdr;
cdrlen = 0;
zeros = 0;
pos = 0;
while (cdrlen < cdrsize) {

        /*
         * Work out which bits to mask out in this iteration.
         */
        last_iter = (cdrlen == cdrsize - 1 && pos != 0);
        switch (pos) {
        case 0:
                /*
                 * High-order 6 bits in current byte.
                 */
                val = cdrp[cdrlen] / P2_2;
                break;
        case 1:
                /*
                 * Low-order 2 bits in current byte
                 * and high-order 4 bits in next byte.
                 */
                val = cdrp[cdrlen] % P2_2 * P2_4;
                if (!last_iter)
                        val += cdrp[cdrlen + 1] / P2_4;
                cdrlen++;
                break;
        case 2:
                /*
                 * Low-order 4 bits in current byte
                 * and high-order 2 bits in next byte.
                 */
                val = cdrp[cdrlen] % P2_4 * P2_2;
                if (!last_iter)
                        val += cdrp[cdrlen + 1] / P2_6;
                cdrlen++;
                break;
        case 3:
                /*
                 * Low-order 6 bits in current byte.
                 */
                val = cdrp[cdrlen] % P2_6;
                cdrlen++;
                break;
        }
        pos = (pos + 1) % 4;

        if (val == 0) {
                /*
                 * val == 0 if a zero block was just peeled off.
                 * If so, increment count of zeros (or output
                 * a run if we have reached the limit of 66).
                 */
                if (zeros == 66) {
                        if ((ior2len += 2) > ior2size)
                                return 0;
                                *ior2++ = ZERO_CODE;
                                *ior2++ = code[zeros - 3];
                                zeros = 1;
                        } else {
                                zeros++;
                        }
                }
                if (val != 0 || last_iter) {
                        /*
                         * We just peeled off a non-zero block, or we
                         * had a zero and this is the last iteration.
                         * Output any zeros found up to this point.
                         * If this is not the last iteration, output
                         * the code for the non-zero block.
                         */
                        switch (zeros) {
                        case 2:
                                if (++ior2len > ior2size)
                                        return 0;
                                *ior2++ = code[0];
                                /* FALLTHROUGH */
                        case 1:
                                if (++ior2len > ior2size)
                                        return 0;
                                *ior2++ = code[0];
                                /* FALLTHROUGH */
                        case 0:
                                if (val != 0) {
                                        if (++ior2len > ior2size)
                                                return 0;
                                        *ior2++ = code[val];
                                }
                                break;
                        default:
                                if ((ior2len += 2) > ior2size)
                                        return 0;
                                *ior2++ = ZERO_CODE;
                                *ior2++ = code[zeros - 3];
                                if (val != 0) {
                                        if (++ior2len > ior2size)
                                                return 0;
                                        *ior2++ = code[val];
                                }
                                break;
                        }
                        zeros = 0;
                }
        }

        /*
         * Write terminating NUL.
         */
        if (++ior2len > ior2size)
                return 0;
        *ior2 = '\0';

        return ior2len;
}

/*--------------------------------------------------------------------------*/
```

```c
/*
 * Convert the reference in 'ior2' into CDR format. The 'cdr' buffer
 * receives the result. 'cdrsize' must be set to the length of the
 * 'cdr' buffer.
 *
 * Return value:
 *
 *       - A non-zero return value indicates the actual number of bytes
 *         written into 'cdr'.
 *
 *       - A zero return value indicates failure:
 *
 *               - either the input IOR did not have an "IOR2" prefix, or
 *
 *               - the ior had a zero length field, or
 *
 *               - 'cdr' was too short to receive the decoded IOR.
 *
 * If the input IOR has a correct "IOR2:" prefix and non-zero length field,
 * it is otherwise assumed to be well-formed. Passing a malformed IOR with
 * an invalid length or illegal characters causes undefined behavior.
 */

size_t
ior2_to_cdr(const char *ior2, void *cdr, size_t cdrsize)
{
        unsigned char   *cdrp;          /* Next byte in CDR */
        unsigned        pos;            /* State variable */
        unsigned        zeros;          /* Num six-bit zeros blocks in run */
        unsigned        val;            /* Decoded value of current IOR2 byte */
        size_t          cdrlen;         /* Num CDR bytes written */
        unsigned        ior2_byte;      /* Current byte of ior2 */
        int             last_iter;      /* True for final 6-bit block */

        /*
         * Initialize code value array (if not done yet).
         */
        if (!once) {
                init_codevals();
                once = 1;
        }

        /*
         * Test prefix.
         */
        if (
               *ior2++ != 'I'
            || *ior2++ != 'O'
            || *ior2++ != 'R'
            || *ior2++ != '2'
            || *ior2++ != ':') {
                return 0;
        }

        /*
         * Test CDR length for sanity.
         */
        if (cdrsize == 0)
                return 0;

        /*
```

```
         * Decode 4-byte CDR length.
         */
        cdrlen = decode[(unsigned)*ior2++] * P2_18;
        cdrlen += decode[(unsigned)*ior2++] * P2_12;
        cdrlen += decode[(unsigned)*ior2++] * P2_6;
        cdrlen += decode[(unsigned)*ior2++];
        if (cdrlen == 0)
                return 0;

        /*
         * Make sure the output buffer is long enough to hold the result.
         */
        if (cdrsize < cdrlen)
                return 0;

        /*
         * Iterate over the IOR2 bytes, decoding each byte into
         * six bits of CDR.
         */
        cdrp = (unsigned char *)cdr;
        pos = 0;
        zeros = 0;
        while (zeros || (ior2_byte = *ior2) != '\0') {
                if (zeros) {
                        /*
                         * val == 0 from previous iteration, do
                         * another zero.
                         */
                        zeros--;
                } else if (ior2_byte == (unsigned)ZERO_CODE) {
                        /*
                         * Deal with zero run.
                         */
                        ior2++;                         /* Skip '=' marker */
                        ior2_byte = *ior2++;            /* Get run length */
                        zeros = decode[ior2_byte] + 2;  /* Count of zeros */
                        val = decode[0];                /* val = 0 */
                } else {
                        /*
                         * Normal decoding.
                         */
                        val = decode[ior2_byte];
                        ior2++;
                }

                /*
                 * Use value of current byte, and copy
                 * bottom six bits of the value
                 * into CDR at the current output position.
                 */
                last_iter = (zeros == 0 && *ior2 == '\0');
                switch (pos) {
                case 0:
                        *cdrp = val * P2_2;
                        break;
                case 1:
                        *cdrp++ += val / P2_4;
                        if (!last_iter)
                                *cdrp = val % P2_4 * P2_4;
                        break;
                case 2:
                        *cdrp++ += val / P2_2;
                        if (!last_iter)
                                *cdrp = val % P2_2 * P2_6;
                        break;
                case 3:
                        *cdrp++ += val;
                        break;
                }
                pos = (pos + 1) % 4;
        }

        return cdrlen;
}

/*--------------------------------------------------------------------------*/

/*
 * Return the number of CDR bytes required to hold a decoded IOR.
 */

size_t
cdr_size(const char *ior)
{
        size_t  len;

        /*
         * Check prefix.
         */
        if (*ior++ != 'I' || *ior++ != 'O' || *ior++ != 'R')
                return 0;

        /*
         * Initialize code value array (if not done yet).
         */
        if (!once) {
                init_codevals();
                once = 1;
        }

        /*
         * Determine IOR version.
         */
        if (*ior == '2' && *(ior + 1) == ':') {
                /*
                 * New IOR, decode length.
                 */
                ior += 2;
                len = decode[(unsigned)*ior++] * P2_18;
                len += decode[(unsigned)*ior++] * P2_12;
                len += decode[(unsigned)*ior++] * P2_6;
                len += decode[(unsigned)*ior];
        } else if (*ior == ':') {
                /*
                 * Old IOR, count number of digits. If a free-standing
                 * implementation is not required, calling strlen() here
                 * will probably be faster.
                 */
                ++ior;
                len = 0;
                while (*ior++ != '\0')
                        len++;
                len /= 2;
        } else {
                /*
                 * Not an "IOR:" or "IOR2:" prefix.
                 */
                len = 0;
        }

        return len;
}
```

8.3 main.c

```
/*
 * Copyright (C) DSTC Pty Ltd (ACN 052 372 577) 1997.
 *
 * Any party obtaining a copy of this file from DSTC Pty Ltd, directly
 * or indirectly, is granted, free of charge, a full and unrestricted
 * irrevocable, world-wide, paid up, royalty-free, nonexclusive right and
 * license to deal in this software (the "Software"), including without
 * limitation the rights to use, copy, modify, merge, publish, distribute,
 * sublicense, and/or sell copies of the Software, and to permit persons
 * who receive copies from any such party to do so, with the only
 * requirement being that all copyright notices remain intact.
 *
 * This software is being provided "AS IS" without warranty of any kind,
 * express, implied or otherwise, including without limitation, any
 * warranty of merchantability or fitness for a particular purpose.
 *
 * In no event shall DSTC Pty Ltd be liable for any special, incidental,
 * indirect or consequential damages of any kind, or any damages whatsoever
 * resulting from loss of use, data or profits, whether or not advised of
 * the possibility of damage, and on any theory of liability, arising out
 * of or in connection with the use or performance of this software.
 *
 * DSTC Pty Ltd welcomes comments and bug fixes related to this software.
 * Please address any queries, comments or bug fixes (please include
 * the name and version of the software you are using and details
 * of your operating system version) to the address below:
 *
 *      DSTC Pty Ltd
 *      Level 7, Gehrmann Labs
 *      University of Queensland
 *      St Lucia, 4072
 *      Tel: +61 7 365 4310
 *      Fax: +61 7 365 4311
 *      Email: Enquiries@dstc.edu.au
 */

/*-------------------------------------------------------------------------*/

/*
 * to_ior, to_ior2 - simple test driver for CDR to IOR2 conversion.
 *
 * Usage: echo <ior> | to_ior2  # Convert normal IOR to IOR2 format.
 *        echo <ior2> | to_ior  # Convert IOR2 format to normal IOR.
 *
 * The conversion direction is determined by looking at argv[0], so
 * the binary must be linked or copied to "to_ior" and "to_ior2" file names.
 *
 * Since this isn't meant to be a serious tool, conversions between normal
 * and IOR2 format are done via intermediate CDR. This is inefficient -
 * for industrial-strength use, it would be better to write direct
 * conversion code.
 * However, this is good enough to demonstrate the conversion from CDR to
 * IOR2 format and back, which is what is required by an ORB run-time.
 */

/*--------------------------------------------------------------------------*/

#include        <stdio.h>
#include        <stdlib.h>
#include        <stdarg.h>
#include        <string.h>
#include        <errno.h>
#include        "ior2.h"

/*--------------------------------------------------------------------------*/

static const char               *progname;
static const char * const       USAGE = "<in_IOR >out_IOR";
static int                      codeval[256];

/*--------------------------------------------------------------------------*/

/*
 * Initialize codeval array for conversion from old IOR format to CDR.
 */

static void
init_codevals()
{
        codeval[(unsigned)'0'] = 0;
        codeval[(unsigned)'1'] = 1;
        codeval[(unsigned)'2'] = 2;
        codeval[(unsigned)'3'] = 3;
        codeval[(unsigned)'4'] = 4;
        codeval[(unsigned)'5'] = 5;
        codeval[(unsigned)'6'] = 6;
        codeval[(unsigned)'7'] = 7;
        codeval[(unsigned)'8'] = 8;
        codeval[(unsigned)'9'] = 9;
        codeval[(unsigned)'a'] = 10;
        codeval[(unsigned)'b'] = 11;
        codeval[(unsigned)'c'] = 12;
        codeval[(unsigned)'d'] = 13;
        codeval[(unsigned)'e'] = 14;
        codeval[(unsigned)'f'] = 15;
        codeval[(unsigned)'A'] = 10;
        codeval[(unsigned)'B'] = 11;
        codeval[(unsigned)'C'] = 12;
        codeval[(unsigned)'D'] = 13;
        codeval[(unsigned)'E'] = 14;
        codeval[(unsigned)'F'] = 15;
}

/*--------------------------------------------------------------------------*/

/*
 * Some systems provide a basename() call as part of libc. However,
 * basename() is not part of XPG4, so there is a version of it below
 * for systems that don't provide it.
 */
```

```
#if     !defined(HAS_BASENAME)

/*
 * Return the last element of the pathname in 'path', deleting any trailing
 * '/' characters. For example, basename("/usr/lib") is "lib", and
 * basename("/usr/") is "usr".
 * If 'path' is NULL, or *path is '\0', a pointer to the constant string "."
 * is returned. Pathnames consisting of only '/' characters return "/".
 *
 * Note: The return value may point into the argument string.
 *       If 'path' has trailing '/' characters, these characters are
 *       overwritten with '\0'.
 */

static char *
basename(char *path)
{
        char    *p;

        /*
         * Check for NULL and empty path.
         */
        if (path == NULL || *path == '\0')
                return(".");

        /*
         * Replace trailing slashes with NULs.
         */
        p = path + strlen(path) - 1;
        while (p >= path && *p == '/')
                *p-- = '\0';

        /*
         * If the path contained only slashes, return "/".
         */
        if (p < path)
                return("/");

        /*
         * Scan from the end for a slash. If no slashes found, just return
         * the path. Otherwise, return what follows the last slash.
         */
        p = strrchr(path, '/');
        if (p == NULL)
                return(path);
        return(p + 1);
}

#endif  /* !defined(HAS_BASENAME) */

/*---------------------------------------------------------------------------*/

/*
 * Print a usage message and exit with status 'st'. If 'st' is zero, print
 * the message on stdout, stderr otherwise.
 */

static void
usage(int st)
{
        (void) fprintf(
                st != 0 ? stderr : stdout, "Usage: %s %s\n", progname, USAGE
        );
        exit(st);
        /* NOTREACHED */
}

/*---------------------------------------------------------------------------*/

/*
 * Print a message on stderr. Prepend the program name and append a
 * newline to the message, then exit.
 */

/* PRINTFLIKE1 */
static void
err(const char *format, ...)
{
        va_list ap;

        va_start(ap, format);
        (void) fprintf(stderr, "%s: ", progname);
        (void) vfprintf(stderr, format, ap);
        (void) putc('\n', stderr);
        va_end(ap);
        exit(EXIT_FAILURE);
        /* NOTREACHED */
}

/*---------------------------------------------------------------------------*/

/*
 * Grow a buffer by doubling its size.
 */

static void
grow_buf(char **buf, size_t *buf_size)
{
        *buf_size *= 2;
        if ((*buf = (char *)realloc((void *)*buf, *buf_size)) == NULL)
                err("Out of memory.");
}

/*---------------------------------------------------------------------------*/

int
main(int argc, char *argv[])
{
        char            *inbuf = NULL;          /* Input buffer */
        size_t          inbuf_limit = 1024;     /* Initial size of inbuf */
        size_t          inbuf_len;              /* Num bytes in inbuf */
        unsigned char   *cdrbuf = NULL;         /* CDR buffer */
        size_t          cdrbuf_len;             /* Size of CDR buffer */
        unsigned char   *cdrp;                  /* Cursor into cdrbuf */
        char            *outbuf = NULL;         /* Output buffer */
        size_t          outbuf_len;             /* Size of outbuf */
        int             c;                      /* Current input char */
        char            *iorp;                  /* Cursor into old IOR */
        unsigned        i;                      /* Counter */
        int             to_ior2;                /* Direction of conversion */
        /*
         * Set program basename for error messages.
         */
        progname = basename(argv[0]);

        /*
         * Check usage.
         */
        if (argc == 2 && strcmp(argv[1], "-?") == 0)
                usage(EXIT_SUCCESS);
        else if (argc != 1)
                usage(EXIT_FAILURE);

        /*
         * Initialize translation table.
         */
        init_codevals();

        /*
         * Figure out conversion direction.
         */
        to_ior2 = strcmp(progname, "to_ior2") == 0;

        /*
         * Create buffer space for input IOR.
         */
        inbuf = (char *)malloc(inbuf_limit);
        if (inbuf == NULL)
                err("Out of memory.");

        /*
         * Read input string.
         */
        errno = 0;
        inbuf_len = 0;
        while ((c = getchar()) != EOF && c != '\n') {
                if (inbuf_len == inbuf_limit)
                        grow_buf(&inbuf, &inbuf_limit);
                inbuf[inbuf_len++] = c;
        }
        inbuf[inbuf_len] = '\0';

        /*
         * May have terminated the read loop early because of an error.
         */
        if (errno != 0)
                err("Error reading input: %s", strerror(errno));

        /*
         * Make buffer for CDR representation.
         */
        if ((cdrbuf_len = cdr_size(inbuf)) == 0)
                err("Malformed input IOR: \"%s\"", inbuf);
        if ((cdrbuf = (unsigned char *)malloc(cdrbuf_len)) == NULL)
                err("Out of memory.");

        /*
         * Convert to CDR.
         */
        if (to_ior2) {
                /*
```

```
                 * Conversion from old IOR to CDR.
                 */
                iorp = inbuf;
                iorp += 4;                      /* Skip IOR: prefix */
                cdrp = cdrbuf;
                cdrbuf_len = 0;
                while (*iorp != '\0' && *(iorp + 1) != '\0') {
                        *cdrp = codeval[(unsigned)*iorp++] * 16;
                        *cdrp++ += codeval[(unsigned)*iorp++] % 16;
                        cdrbuf_len++;
                }

                /*
                 * For conversion from old IOR format to IOR2, we allocate
                 * the same space as is occupied by the old IOR. Since
                 * IOR2 format is shorter, this will do.
                 */
                outbuf_len = inbuf_len;
        } else {
                /*
                 * Conversion from IOR2 to CDR.
                 */
                if (ior2_to_cdr(inbuf, (void *)cdrbuf, cdrbuf_len) == 0)
                        err("Cannot convert IOR2 to CDR.");

                /*
                 * For conversion from IOR2 to old IOR format, we need
                 * 2 bytes for each CDR byte, plus 5 bytes for the prefix,
                 * plus room for the terminating NUL.
                 */
                outbuf_len = cdrbuf_len * 2 + 5 + 1;
        }

        /*
         * Allocate output buffer space.
         */
        if ((outbuf = (char *)malloc(outbuf_len)) == NULL)
                err("Out of memory.");

        /*
         * Convert from CDR to output format.
         */
        if (to_ior2) {
                outbuf_len = cdr_to_ior2(
                        (void *)cdrbuf, cdrbuf_len, outbuf, outbuf_len
                );
                if (outbuf_len == 0)
                        err("Cannot convert from CDR to IOR2.");
        } else {
                iorp = outbuf;
                *iorp++ = 'I';
                *iorp++ = 'O';
                *iorp++ = 'R';
                *iorp++ = ':';
                for (i = 0; i != cdrbuf_len; i++) {
                        (void) sprintf(iorp, "%02X", cdrbuf[i]);
                        iorp += 2;
                }
        }

        /*
```

```
	 * Print output IOR.
	 */
	if (fputs(outbuf, stdout) == EOF)
		err("Error writing output: %s", strerror(errno));
	if (fputc('\n', stdout) == EOF)
		err("Error writing output: %s", strerror(errno));
	if (fclose(stdout) == EOF)
		err("Error closing stdout: %s", strerror(errno));

	/*
	 * Clean up.
	 */
	free((void *)outbuf);
	free((void *)cdrbuf);
	free((void *)inbuf);

	return EXIT_SUCCESS;
}
```