```
**********************************************************
   26894 Sat Apr 20 02:07:53 2013
new/usr/src/uts/common/fs/zfs/lz4.c
Integrated r91 LZ4.
**********************************************************
```

```
/*
 * LZ4 - Fast LZ compression algorithm
 * Header File
 * Copyright (C) 2011-2013, Yann Collet.
 * BSD 2-Clause License (http://www.opensource.org/licenses/bsd-license.php)
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 *     * Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *     * Redistributions in binary form must reproduce the above
 * copyright notice, this list of conditions and the following disclaimer
 * in the documentation and/or other materials provided with the
 * distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * You can contact the author at :
 * - LZ4 homepage : http://fastcompression.blogspot.com/p/lz4.html
 * - LZ4 source repository : http://code.google.com/p/lz4/
 * Upstream release : r91
 */

#include <sys/zfs_context.h>

static int real_LZ4_compress(const char *source, char *dest, int isize,
    int osize);
static int real_LZ4_uncompress(const char *source, char *dest, int osize);
static int LZ4_compressBound(int isize);
static int LZ4_uncompress_unknownOutputSize(const char *source, char *dest,
    int isize, int maxOutputSize);
static int LZ4_compressCtx(void *ctx, const char *source, char *dest,
    int isize, int osize);
static int LZ4_compress64kCtx(void *ctx, const char *source, char *dest,
    int isize, int osize);

/*ARGSUSED*/
size_t
lz4_compress(void *s_start, void *d_start, size_t s_len, size_t d_len, int n)
{
        uint32_t bufsiz;
        char *dest = d_start;

        ASSERT(d_len >= sizeof (bufsiz));

        bufsiz = real_LZ4_compress(s_start, &dest[sizeof (bufsiz)], s_len,
            d_len - sizeof (bufsiz));

        /* Signal an error if the compression routine returned zero. */
```



```
        if (bufsiz == 0)
                return (s_len);

        /*
         * Encode the compresed buffer size at the start. We'll need this in
         * decompression to counter the effects of padding which might be
         * added to the compressed buffer and which, if unhandled, would
         * confuse the hell out of our decompression function.
         */
        *(uint32_t *)dest = BE_32(bufsiz);

        return (bufsiz + sizeof (bufsiz));
}
```

*unchanged_portion_omitted_*

```
/*
 * LZ4 API Description:
 *
 * Simple Functions:
 * real_LZ4_compress() :
 *      isize  : is the input size. Max supported value is ~1.9GB
 *      return : the number of bytes written in buffer dest
 *               or 0 if the compression fails (if LZ4_COMPRESSMIN is set).
 *      note : destination buffer must be already allocated.
 *              destination buffer must be sized to handle worst cases
 *              situations (input data not compressible) worst case size
 *              evaluation is provided by function LZ4_compressBound().
 *
 * real_LZ4_uncompress() :
 *      osize  : is the output size, therefore the original size
 *      return : the number of bytes read in the source buffer.
 *              If the source stream is malformed, the function will stop
 *              decoding and return a negative result, indicating the byte
 *              position of the faulty instruction. This function never
 *              writes beyond dest + osize, and is therefore protected
 *              against malicious data packets.
 *      note : destination buffer must be already allocated
 *
 * Advanced Functions
 *
 * LZ4_compressBound() :
 *      Provides the maximum size that LZ4 may output in a "worst case"
 *      scenario (input data not compressible) primarily useful for memory
 *      allocation of output buffer.
 *
 *      isize  : is the input size. Max supported value is ~1.9GB
 *      return : maximum output size in a "worst case" scenario
 *      note : this function is limited by "int" range (2^31-1)
 *
 * LZ4_uncompress_unknownOutputSize() :
 *      isize  : is the input size, therefore the compressed size
 *      maxOutputSize : is the size of the destination buffer (which must be
 *              already allocated)
 *      return : the number of bytes decoded in the destination buffer
 *              (necessarily <= maxOutputSize). If the source stream is
 *              malformed, the function will stop decoding and return a
 *              negative result, indicating the byte position of the faulty
 *              instruction. This function never writes beyond dest +
 *              maxOutputSize, and is therefore protected against malicious
 *              data packets.
 *      note   : Destination buffer must be already allocated.
 *              This version is slightly slower than real_LZ4_uncompress()
 *
 * LZ4_compressCtx() :
 *      This function explicitly handles the CTX memory structure.
 *
```

```
 134  *      ILLUMOS CHANGES: the CTX memory structure must be explicitly allocated
 135  *      by the caller (either on the stack or using kmem_zalloc). Passing NULL
 136  *      isn't valid.
 137  *
 138  * LZ4_compress64kCtx() :
 139  *      Same as LZ4_compressCtx(), but specific to small inputs (<64KB).
 140  *      isize *Must* be <64KB, otherwise the output will be corrupted.
 141  *
 142  *      ILLUMOS CHANGES: the CTX memory structure must be explicitly allocated
 143  *      by the caller (either on the stack or using kmem_zalloc). Passing NULL
 144  *      isn't valid.
 145  */

 147 /*
 148  * Tuning parameters
 149  */

 151 /*
 152  * COMPRESSIONLEVEL: Increasing this value improves compression ratio
 153  *       Lowering this value reduces memory usage. Reduced memory usage
 154  *      typically improves speed, due to cache effect (ex: L1 32KB for Intel,
 155  *      L1 64KB for AMD). Memory usage formula : N->2^(N+2) Bytes
 156  *      (examples : 12 -> 16KB ; 17 -> 512KB)
 157  */
 158 #define COMPRESSIONLEVEL 12

 160 /*
 161  * NOTCOMPRESSIBLE_CONFIRMATION: Decreasing this value will make the
 162  *      algorithm skip faster data segments considered "incompressible".
 163  *      This may decrease compression ratio dramatically, but will be
 164  *      faster on incompressible data. Increasing this value will make
 165  *      the algorithm search more before declaring a segment "incompressible".
 166  *      This could improve compression a bit, but will be slower on
 167  *      incompressible data. The default value (6) is recommended.
 168  */
 169 #define NOTCOMPRESSIBLE_CONFIRMATION 6

 171 /*
 172  * BIG_ENDIAN_NATIVE_BUT_INCOMPATIBLE: This will provide a boost to
 173  * performance for big endian cpu, but the resulting compressed stream
 174  * will be incompatible with little-endian CPU. You can set this option
 175  * to 1 in situations where data will stay within closed environment.
 176  * This option is useless on Little_Endian CPU (such as x86).
 177  */
 178 /* #define      BIG_ENDIAN_NATIVE_BUT_INCOMPATIBLE 1 */

 180 /*
 181  * CPU Feature Detection
 182  */

 184 /* 32 or 64 bits ? */
 185 #if (defined(__x86_64__) || defined(__x86_64) || defined(__amd64__) || \
 186     defined(__amd64) || defined(__ppc64__) || defined(_WIN64) || \
 187     defined(__LP64__) || defined(_LP64))
 188 #define LZ4_ARCH64 1
 189 #else
 190 #define LZ4_ARCH64 0
 191 #endif

 193 /*
 194  * Limits the amount of stack space that the algorithm may consume to hold
 195  * the compression lookup table. The value '9' here means we'll never use
 196  * more than 2k of stack (see above for a description of COMPRESSIONLEVEL).
 197  * If more memory is needed, it is allocated from the heap.
 198  */
 199 #define STACKLIMIT 9
```



```
 201 /*
 202  * Little Endian or Big Endian?
 203  * Note: overwrite the below #define if you know your architecture endianess.
 204  */
 205 #if (defined(__BIG_ENDIAN__) || defined(__BIG_ENDIAN) || \
 206         defined(_BIG_ENDIAN) || defined(_ARCH_PPC) || defined(__PPC__) || \
 207         defined(__PPC) || defined(PPC) || defined(__powerpc__) || \
 208         defined(__powerpc) || defined(powerpc) || \
 209         ((defined(__BYTE_ORDER__)&&(__BYTE_ORDER__ == __ORDER_BIG_ENDIAN__))))
 210 #define LZ4_BIG_ENDIAN 1
 211 #else
 212 /*
 213  * Little Endian assumed. PDP Endian and other very rare endian format
 214  * are unsupported.
 215  */
 216 #endif

 218 /*
 219  * Unaligned memory access is automatically enabled for "common" CPU,
 220  * such as x86. For others CPU, the compiler will be more cautious, and
 221  * insert extra code to ensure aligned access is respected. If you know
 222  * your target CPU supports unaligned memory access, you may want to
 223  * force this option manually to improve performance
 224  */
 225 #if defined(__ARM_FEATURE_UNALIGNED)
 226 #define LZ4_FORCE_UNALIGNED_ACCESS 1
 227 #endif

 229 /* #define      LZ4_FORCE_SW_BITCOUNT */

 231 /*
 232  * Compiler Options
 233  */
 234 #if __STDC_VERSION__ >= 199901L /* C99 */
 235 /* "restrict" is a known keyword */
 236 #else
 237 /* Disable restrict */
 238 #define restrict
 239 #endif

 241 #define GCC_VERSION (__GNUC__ * 100 + __GNUC_MINOR__)

 243 #ifdef _MSC_VER
 244 /* Visual Studio */
 245 /* Visual is not C99, but supports some kind of inline */
 246 #define inline __forceinline
 247 #if LZ4_ARCH64
 248 /* For Visual 2005 */
 249 #pragma intrinsic(_BitScanForward64)
 250 #pragma intrinsic(_BitScanReverse64)
 251 #else /* !LZ4_ARCH64 */
 252 /* For Visual 2005 */
 253 #pragma intrinsic(_BitScanForward)
 254 #pragma intrinsic(_BitScanReverse)
 255 #endif /* !LZ4_ARCH64 */
 256 #endif /* _MSC_VER */

 258 #ifdef _MSC_VER
 259 #define lz4_bswap16(x) _byteswap_ushort(x)
 260 #else /* !_MSC_VER */
 261 #define lz4_bswap16(x) ((unsigned short int) ((((x) >> 8) & 0xffu) | \
 262         (((x) & 0xffu) << 8)))
 263 #endif /* !_MSC_VER */

 265 #if (GCC_VERSION >= 302) || (__INTEL_COMPILER >= 800) || defined(__clang__)
```

```
266 #define expect(expr, value)    (__builtin_expect((expr), (value)))
267 #else
268 #define expect(expr, value)    (expr)
269 #endif

271 #define likely(expr)    expect((expr) != 0, 1)
272 #define unlikely(expr)  expect((expr) != 0, 0)

274 /* Basic types */
275 #if defined(_MSC_VER)
276 /* Visual Studio does not support 'stdint' natively */
277 #define BYTE    unsigned __int8
278 #define U16     unsigned __int16
279 #define U32     unsigned __int32
280 #define S32     __int32
281 #define U64     unsigned __int64
282 #else /* !defined(_MSC_VER) */
283 #define BYTE    uint8_t
284 #define U16     uint16_t
285 #define U32     uint32_t
286 #define S32     int32_t
287 #define U64     uint64_t
288 #endif /* !defined(_MSC_VER) */

290 #ifndef LZ4_FORCE_UNALIGNED_ACCESS
291 #pragma pack(1)
292 #endif

294 typedef struct _U16_S {
295         U16 v;
296 } U16_S;
_____unchanged_portion_omitted_

500 /* Compression functions */

502 /*ARGSUSED*/
503 static int
504 LZ4_compressCtx(void *ctx, const char *source, char *dest, int isize,
505     int osize)
506 {
507 #if HEAPMODE
508         struct refTables *srt = (struct refTables *)ctx;
509         HTYPE *HashTable = (HTYPE *) (srt->hashTable);
510 #else
511         HTYPE HashTable[HASHTABLESIZE] = { 0 };
512 #endif

514         const BYTE *ip = (BYTE *) source;
515         INITBASE(base);
516         const BYTE *anchor = ip;
517         const BYTE *const iend = ip + isize;
518         const BYTE *const oend = (BYTE *) dest + osize;
519         const BYTE *const mflimit = iend - MFLIMIT;
520 #define matchlimit (iend - LASTLITERALS)

522         BYTE *op = (BYTE *) dest;

524         int length;
535         int len, length;
525         const int skipStrength = SKIPSTRENGTH;
526         U32 forwardH;


529         /* Init */
530         if (isize < MINLENGTH)
531                 goto _last_literals;
```



```
533         /* First Byte */
534         HashTable[LZ4_HASH_VALUE(ip)] = ip - base;
535         ip++;
536         forwardH = LZ4_HASH_VALUE(ip);

538         /* Main Loop */
539         for (;;) {
540                 int findMatchAttempts = (1U << skipStrength) + 3;
541                 const BYTE *forwardIp = ip;
542                 const BYTE *ref;
543                 BYTE *token;

545                 /* Find a match */
546                 do {
547                         U32 h = forwardH;
548                         int step = findMatchAttempts++ >> skipStrength;
549                         ip = forwardIp;
550                         forwardIp = ip + step;

552                         if unlikely(forwardIp > mflimit) {
553                                 goto _last_literals;
554                         }

556                         forwardH = LZ4_HASH_VALUE(forwardIp);
557                         ref = base + HashTable[h];
558                         HashTable[h] = ip - base;

560                 } while ((ref < ip - MAX_DISTANCE) || (A32(ref) != A32(ip)));

562                 /* Catch up */
563                 while ((ip > anchor) && (ref > (BYTE *) source) &&
564                     unlikely(ip[-1] == ref[-1])) {
565                         ip--;
566                         ref--;
567                 }

569                 /* Encode Literal length */
570                 length = ip - anchor;
571                 token = op++;

573                 /* Check output limit */
574                 if unlikely(op + length + (2 + 1 + LASTLITERALS) +
575                     (length >> 8) > oend)
576                         return (0);

578                 if (length >= (int)RUN_MASK) {
579                         int len;
580                         *token = (RUN_MASK << ML_BITS);
581                         len = length - RUN_MASK;
582                         for (; len > 254; len -= 255)
583                                 *op++ = 255;
584                         *op++ = (BYTE)len;
585                 } else
586                         *token = (length << ML_BITS);

588                 /* Copy Literals */
589                 LZ4_BLINDCOPY(anchor, op, length);

591                 _next_match:
592                 /* Encode Offset */
593                 LZ4_WRITE_LITTLEENDIAN_16(op, ip - ref);

595                 /* Start Counting */
596                 ip += MINMATCH;
597                 ref += MINMATCH;        /* MinMatch already verified */
```

```
607                 ref += MINMATCH;        /* MinMatch verified */
598                 anchor = ip;
599                 while likely(ip < matchlimit - (STEPSIZE - 1)) {
600                         UARCH diff = AARCH(ref) ^ AARCH(ip);
601                         if (!diff) {
602                                 ip += STEPSIZE;
603                                 ref += STEPSIZE;
604                                 continue;
605                         }
606                         ip += LZ4_NbCommonBytes(diff);
607                         goto _endCount;
608                 }
609 #if LZ4_ARCH64
610                 if ((ip < (matchlimit - 3)) && (A32(ref) == A32(ip))) {
611                         ip += 4;
612                         ref += 4;
613                 }
614 #endif
615                 if ((ip < (matchlimit - 1)) && (A16(ref) == A16(ip))) {
616                         ip += 2;
617                         ref += 2;
618                 }
619                 if ((ip < matchlimit) && (*ref == *ip))
620                         ip++;
621                 _endCount:

623                 /* Encode MatchLength */
624                 length = (int)(ip - anchor);
634                 len = (ip - anchor);
625                 /* Check output limit */
626                 if unlikely(op + (1 + LASTLITERALS) + (length >> 8) > oend)
636                 if unlikely(op + (1 + LASTLITERALS) + (len >> 8) > oend)
627                         return (0);
628                 if (length >= (int)ML_MASK) {
638                 if (len >= (int)ML_MASK) {
629                         *token += ML_MASK;
630                         length -= ML_MASK;
631                         for (; length > 509; length -= 510) {
640                         len -= ML_MASK;
641                         for (; len > 509; len -= 510) {
632                                 *op++ = 255;
633                                 *op++ = 255;
634                         }
635                         if (length > 254) {
636                                 length -= 255;
645                         if (len > 254) {
646                                 len -= 255;
637                                 *op++ = 255;
638                         }
639                         *op++ = (BYTE)length;
649                         *op++ = (BYTE)len;
640                 } else
641                         *token += length;
651                         *token += len;

643                 /* Test end of chunk */
644                 if (ip > mflimit) {
645                         anchor = ip;
646                         break;
647                 }
648                 /* Fill table */
649                 HashTable[LZ4_HASH_VALUE(ip - 2)] = ip - 2 - base;

651                 /* Test next position */
652                 ref = base + HashTable[LZ4_HASH_VALUE(ip)];
653                 HashTable[LZ4_HASH_VALUE(ip)] = ip - base;
```



```
654                 if ((ref > ip - (MAX_DISTANCE + 1)) && (A32(ref) == A32(ip))) {
655                         token = op++;
656                         *token = 0;
657                         goto _next_match;
658                 }
659                 /* Prepare next loop */
660                 anchor = ip++;
661                 forwardH = LZ4_HASH_VALUE(ip);
662         }

664         _last_literals:
665         /* Encode Last Literals */
666         {
667                 int lastRun = iend - anchor;
668                 if (op + lastRun + 1 + ((lastRun + 255 - RUN_MASK) / 255) >
669                     oend)
670                         return (0);
671                 if (lastRun >= (int)RUN_MASK) {
672                         *op++ = (RUN_MASK << ML_BITS);
673                         lastRun -= RUN_MASK;
674                         for (; lastRun > 254; lastRun -= 255) {
675                                 *op++ = 255;
676                         }
677                         *op++ = (BYTE)lastRun;
678                 } else
679                         *op++ = (lastRun << ML_BITS);
680                 (void) memcpy(op, anchor, iend - anchor);
681                 op += iend - anchor;
682         }

684         /* End */
685         return (int)(((char *)op) - dest);
686 }
______unchanged_portion_omitted_

908 /* Decompression functions */

920 /*
921  * Note: The decoding functions real_LZ4_uncompress() and
922  *      LZ4_uncompress_unknownOutputSize() are safe against "buffer overflow"
923  *      attack type. They will never write nor read outside of the provided
924  *      output buffers. LZ4_uncompress_unknownOutputSize() also insures that
925  *      it will never read outside of the input buffer. A corrupted input
926  *      will produce an error result, a negative int, indicating the position
927  *      of the error within input stream.
928  */

910 static int
911 LZ4_uncompress_unknownOutputSize(const char *source, char *dest, int isize,
912     int maxOutputSize)
931 real_LZ4_uncompress(const char *source, char *dest, int osize)
913 {
914         /* Local Variables */
915         const BYTE *restrict ip = (const BYTE *) source;
916         const BYTE *const iend = ip + isize;
917         const BYTE *ref;

919         BYTE *op = (BYTE *) dest;
920         BYTE *const oend = op + maxOutputSize;
938         BYTE *const oend = op + osize;
921         BYTE *cpy;

941         unsigned token;

943         size_t length;
923         size_t dec32table[] = {0, 3, 2, 3, 0, 0, 0, 0};
```

```
#if LZ4_ARCH64
        size_t dec64table[] = {0, 0, 0, (size_t)-1, 0, 1, 2, 3};
#endif

        /* Main Loop */
        for (;;) {
                /* get runlength */
                token = *ip++;
                if ((length = (token >> ML_BITS)) == RUN_MASK) {
                        size_t len;
                        for (; (len = *ip++) == 255; length += 255) {
                        }
                        length += len;
                }
                /* copy literals */
                cpy = op + length;
                if unlikely(cpy > oend - COPYLENGTH) {
                        if (cpy != oend)
                                /* Error: we must necessarily stand at EOF */
                                goto _output_error;
                        (void) memcpy(op, ip, length);
                        ip += length;
                        break;  /* EOF */
                        }
                LZ4_WILDCOPY(ip, op, cpy);
                ip -= (op - cpy);
                op = cpy;

                /* get offset */
                LZ4_READ_LITTLEENDIAN_16(ref, cpy, ip);
                ip += 2;
                if unlikely(ref < (BYTE * const) dest)
        /*
         * Special case
         * A correctly formed null-compressed LZ4 must have at least
         * one byte (token=0)
                         * Error: offset create reference outside destination
                         * buffer
         */
        if (unlikely(ip == iend))
                goto _output_error;

                /* get matchlength */
                if ((length = (token & ML_MASK)) == ML_MASK) {
                        for (; *ip == 255; length += 255) {
                                ip++;
                        }
                        length += *ip++;
                }
                /* copy repeated sequence */
                if unlikely(op - ref < STEPSIZE) {
#if LZ4_ARCH64
                        size_t dec64 = dec64table[op-ref];
#else
                        const int dec64 = 0;
#endif
                        op[0] = ref[0];
                        op[1] = ref[1];
                        op[2] = ref[2];
                        op[3] = ref[3];
                        op += 4;
                        ref += 4;
                        ref -= dec32table[op-ref];
                        A32(op) = A32(ref);
                        op += STEPSIZE - 4;
                        ref -= dec64;
```



```
                } else {
                        LZ4_COPYSTEP(ref, op);
                }
                cpy = op + length - (STEPSIZE - 4);
                if (cpy > oend - COPYLENGTH) {
                        if (cpy > oend)
                                /*
                                 * Error: request to write beyond destination
                                 * buffer
                                 */
                                goto _output_error;
                        LZ4_SECURECOPY(ref, op, (oend - COPYLENGTH));
                        while (op < cpy)
                                *op++ = *ref++;
                        op = cpy;
                        if (op == oend)
                                /*
                                 * Check EOF (should never happen, since last
                                 * 5 bytes are supposed to be literals)
                                 */
                                goto _output_error;
                        continue;
                }
                LZ4_SECURECOPY(ref, op, cpy);
                op = cpy;       /* correction */
        }

        /* end of decoding */
        return (int)(((char *)ip) - source);

        /* write overflow error detected */
        _output_error:
        return (int)(-(((char *)ip) - source));
}

static int
LZ4_uncompress_unknownOutputSize(const char *source, char *dest, int isize,
    int maxOutputSize)
{
        /* Local Variables */
        const BYTE *restrict ip = (const BYTE *) source;
        const BYTE *const iend = ip + isize;
        const BYTE *ref;

        BYTE *op = (BYTE *) dest;
        BYTE *const oend = op + maxOutputSize;
        BYTE *cpy;

        size_t dec32table[] = {0, 3, 2, 3, 0, 0, 0, 0};
#if LZ4_ARCH64
        size_t dec64table[] = {0, 0, 0, (size_t)-1, 0, 1, 2, 3};
#endif

        /* Main Loop */
        /*LINTED E_CONSTANT_CONDITION*/
        while (1) {
        while (ip < iend) {
                unsigned token;
                size_t length;

                /* get runlength */
                token = *ip++;
                if ((length = (token >> ML_BITS)) == RUN_MASK) {
                        int s = 255;
                        while ((ip < iend) && (s == 255)) {
                                s = *ip++;
```

```
 948                                length += s;
 949                        }
 950                }
 951                /* copy literals */
 952                cpy = op + length;
 953                if ((cpy > oend - MFLIMIT) ||
 954                    (ip + length > iend - (2 + 1 + LASTLITERALS))) {
1076                if ((cpy > oend - COPYLENGTH) ||
1077                    (ip + length > iend - COPYLENGTH)) {
 955                        if (cpy > oend)
 956                                /* Error: writes beyond output buffer */
 957                                goto _output_error;
 958                        if (ip + length != iend)
 959                                /*
 960                                 * Error: LZ4 format requires to consume all
 961                                 * input at this stage (no match within the
 962                                 * last 11 bytes, and at least 8 remaining
 963                                 * input bytes for another match + literals
1084                                 * input at this stage
 964                                 */
 965                                goto _output_error;
 966                        (void) memcpy(op, ip, length);
 967                        op += length;
 968                        /* Necessarily EOF, due to parsing restrictions */
 969                        break;
 970                }
 971                LZ4_WILDCOPY(ip, op, cpy);
 972                ip -= (op - cpy);
 973                op = cpy;

 975                /* get offset */
 976                LZ4_READ_LITTLEENDIAN_16(ref, cpy, ip);
 977                ip += 2;
 978                if (unlikely(ref < (BYTE * const) dest))
1099                if (ref < (BYTE * const) dest)
 979                        /*
 980                         * Error: offset creates reference outside of
 981                         * destination buffer
 982                         */
 983                        goto _output_error;

 985                /* get matchlength */
 986                if ((length = (token & ML_MASK)) == ML_MASK) {
 987                        while (likely(ip < iend - (LASTLITERALS + 1))) {
1108                        while (ip < iend) {
 988                                int s = *ip++;
 989                                length += s;
 990                                if (s == 255)
 991                                        continue;
 992                                break;
 993                        }
 994                }
 995                /* copy repeated sequence */
 996                if unlikely(op - ref < STEPSIZE) {
 997 #if LZ4_ARCH64
 998                        size_t dec64 = dec64table[op-ref];
 999 #else
1000                        const int dec64 = 0;
1001 #endif
1002                        op[0] = ref[0];
1003                        op[1] = ref[1];
1004                        op[2] = ref[2];
1005                        op[3] = ref[3];
1006                        op += 4;
1007                        ref += 4;
1008                        ref -= dec32table[op-ref];
```



```
1009                        A32(op) = A32(ref);
1010                        op += STEPSIZE - 4;
1011                        ref -= dec64;
1012                } else {
1013                        LZ4_COPYSTEP(ref, op);
1014                }
1015                cpy = op + length - (STEPSIZE - 4);
1016                if (unlikely(cpy > oend - (COPYLENGTH + (STEPSIZE - 4)))) {
1017                        if (cpy > oend - LASTLITERALS)
1137                if (cpy > oend - COPYLENGTH) {
1138                        if (cpy > oend)
1018                                /*
1019                                 * Error: last 5 bytes must be literals
1140                                 * Error: request to write outside of
1141                                 * destination buffer
1020                                 */
1021                                goto _output_error;
1022                        LZ4_SECURECOPY(ref, op, (oend - COPYLENGTH));
1023                        while (op < cpy)
1024                                *op++ = *ref++;
1025                        op = cpy;
1026                        if (op == oend)
1027                                /*
1028                                 * Check EOF (should never happen, since
1029                                 * last 5 bytes are supposed to be literals)
1030                                 */
1031                                goto _output_error;
1032                        continue;
1033                }
1034                LZ4_WILDCOPY(ref, op, cpy);
1156                LZ4_SECURECOPY(ref, op, cpy);
1035                op = cpy;       /* correction */
1036        }

1038        /* end of decoding */
1039        return (int)(((char *)op) - dest);

1041        /* write overflow error detected */
1042        _output_error:
1043        return (int)(-(((char *)ip) - source));
1044 }
```

_____***unchanged_portion_omitted_***