```
**********************************************************
    1596 Wed Feb 25 22:19:02 2009
new/usr/src/common/bignum/README
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
# Common Development and Distribution License, Version 1.0 only
# (the "License").  You may not use this file except in compliance
# with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#ident  "%Z%%M% %I%     %E% SMI"

A note on the file mont_mulf.c.
A note on the file mont_mulf.c. It is used as a starting point
for the following hand optimized assembly files.

It is used as a starting point for the following hand optimized assembly files:

- usr/src/common/bignum/sun4u/mont_mulf_v8plus.s
 This file is used in the 32-bit pkcs11_softtoken library.

- usr/src/common/bignum/sun4u/mont_mulf_v9.s
 This file is used in the 64-bit pkcs11_softtoken library.

- usr/src/common/bignum/sun4u/mont_mulf_kernel_v9.s
- usr/src/uts/sun4u/rsa/mont_mulf.s
 This file is used in the kernel RSA module.

We keep the .c file around for future reference.

See file bignumimpl.c for information about these preprocessor symbols:
        USE_FLOATING_POINT, PSR_MUL, HWCAP, UMUL64
UMUL64 is set in bignum.h based on architecture.
The other symbols are set in Makefiles, as appropriate.
```

```
**********************************************************
    1834 Wed Feb 25 22:19:15 2009
new/usr/src/common/bignum/amd64/bignum_amd64.c
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * This file contains bignum implementation code that
 * is specific to AMD64, but which is still more appropriate
 * to write in C, rather than assembly language.
 * bignum_amd64_asm.s does all the assembly language code
 * for AMD64 specific bignum support.  The assembly language
 * source file has pure code, no data.  Let the C compiler
 * generate what is needed to handle the variations in
 * data representation and addressing, for example,
 * statically linked vs PIC.
 */

#include "bignum.h"

/*
 * The bignum interface deals with arrays of 64-bit "chunks" or "digits".
 * Data should be aligned on 8-byte address boundaries for best performance.
 * The bignum interface deals only with arrays of 32-bit "digits".
 * The 64-bit bignum functions are internal implementation details.
 * If a bignum happens to be aligned on a 64-bit boundary
 * and its length is even, then the pure 64-bit implementation
 * can be used.
 */

#define ISALIGNED64(p) (((uintptr_t)(p) & 7) == 0)
#define ISBIGNUM64(p, len) (ISALIGNED64(p) && (((len) & 1) == 0))

#if defined(__lint)
```



```
extern uint64_t *P64(uint32_t *addr);

#else /* lint */

#define P64(addr) ((uint64_t *)addr)

#endif /* lint */

extern uint64_t big_mul_set_vec64(uint64_t *, uint64_t *, int, uint64_t);
extern uint64_t big_mul_add_vec64(uint64_t *, uint64_t *, int, uint64_t);
extern void big_mul_vec64(uint64_t *, uint64_t *, int, uint64_t *, int);
extern void big_sqr_vec64(uint64_t *, uint64_t *, int);

extern uint32_t big_mul_set_vec32(uint32_t *, uint32_t *, int, uint32_t);
extern uint32_t big_mul_add_vec32(uint32_t *, uint32_t *, int, uint32_t);
extern void big_mul_vec32(uint32_t *, uint32_t *, int, uint32_t *, int);
extern void big_sqr_vec32(uint32_t *, uint32_t *, int);

uint32_t big_mul_set_vec(uint32_t *, uint32_t *, int, uint32_t);
uint32_t big_mul_add_vec(uint32_t *, uint32_t *, int, uint32_t);
void big_mul_vec(uint32_t *, uint32_t *, int, uint32_t *, int);
void big_sqr_vec(uint32_t *, uint32_t *, int);


void
big_mul_vec(BIG_CHUNK_TYPE *r, BIG_CHUNK_TYPE *a, int alen,
    BIG_CHUNK_TYPE *b, int blen)
big_mul_vec(uint32_t *r, uint32_t *a, int alen, uint32_t *b, int blen)
{
        if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen) || !ISBIGNUM64(b, blen)) {
                big_mul_vec32(r, a, alen, b, blen);
                return;
        }

        big_mul_vec64(P64(r), P64(a), alen / 2, P64(b), blen / 2);
}

void
big_sqr_vec(uint32_t *r, uint32_t *a, int alen)
{
        if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen)) {
                big_mul_vec32(r, a, alen, a, alen);
                return;
        }
        big_sqr_vec64(P64(r), P64(a), alen / 2);
}

/*
 * It is OK to cast the 64-bit carry to 32 bit.
 * There will be no loss, because although we are multiplying the vector, a,
 * by a uint64_t, its value cannot exceedthat of a uint32_t.
 */

uint32_t
big_mul_set_vec(uint32_t *r, uint32_t *a, int alen, uint32_t digit)
{
        if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen))
                return (big_mul_set_vec32(r, a, alen, digit));

        return (big_mul_set_vec64(P64(r), P64(a), alen / 2, digit));
}
uint32_t
big_mul_add_vec(uint32_t *r, uint32_t *a, int alen, uint32_t digit)
{
        if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen))
```

```
119                 return (big_mul_add_vec32(r, a, alen, digit));

121         return (big_mul_add_vec64(P64(r), P64(a), alen / 2, digit));
122 }


125 void
126 big_mul_vec64(uint64_t *r, uint64_t *a, int alen, uint64_t *b, int blen)
127 {
 50         int     i;

 52         r[alen] = big_mul_set_vec(r, a, alen, b[0]);
130         r[alen] = big_mul_set_vec64(r, a, alen, b[0]);
 53         for (i = 1; i < blen; ++i)
 54                 r[alen + i] = big_mul_add_vec(r + i, a, alen, b[i]);
132                 r[alen + i] = big_mul_add_vec64(r+i, a, alen, b[i]);
133 }

135 void
136 big_mul_vec32(uint32_t *r, uint32_t *a, int alen, uint32_t *b, int blen)
137 {
138         int i;

140         r[alen] = big_mul_set_vec32(r, a, alen, b[0]);
141         for (i = 1; i < blen; ++i)
142                 r[alen + i] = big_mul_add_vec32(r+i, a, alen, b[i]);
143 }

145 void
146 big_sqr_vec32(uint32_t *r, uint32_t *a, int alen)
147 {
148         big_mul_vec32(r, a, alen, a, alen);
149 }


152 uint32_t
153 big_mul_set_vec32(uint32_t *r, uint32_t *a, int alen, uint32_t digit)
154 {
155         uint64_t p, d, cy;

157         d = (uint64_t)digit;
158         cy = 0;
159         while (alen != 0) {
160                 p = (uint64_t)a[0] * d + cy;
161                 r[0] = (uint32_t)p;
162                 cy = p >> 32;
163                 ++r;
164                 ++a;
165                 --alen;
166         }
167         return ((uint32_t)cy);
168 }

170 uint32_t
171 big_mul_add_vec32(uint32_t *r, uint32_t *a, int alen, uint32_t digit)
172 {
173         uint64_t p, d, cy;

175         d = (uint64_t)digit;
176         cy = 0;
177         while (alen != 0) {
178                 p = r[0] + (uint64_t)a[0] * d + cy;
179                 r[0] = (uint32_t)p;
180                 cy = p >> 32;
181                 ++r;
182                 ++a;
```



```
183                 --alen;
184         }
185         return ((uint32_t)cy);
 55 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   12816 Wed Feb 25 22:19:27 2009
new/usr/src/common/bignum/amd64/bignum_amd64_asm.s
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2004 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <sys/asm_linkage.h>

#if defined(lint) || defined(__lint)

#include <sys/types.h>

/* ARGSUSED */
uint64_t
big_mul_set_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
{ return (0); }

/* ARGSUSED */
uint64_t
big_mul_add_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
{ return (0); }

/* ARGSUSED */
void
big_sqr_vec(uint64_t *r, uint64_t *a, int len)
{}

#else   /* lint */

/ ---------------------------------------------------------------------------
/
```



```
/  Implementation of big_mul_set_vec which exploits
/  the 64X64->128 bit  unsigned multiply instruction.
/
/  As defined in Sun's bignum library for pkcs11, bignums are
/  composed of an array of 64-bit "digits" or "chunks" along with
/  descriptive information.
/  composed of an array of 32-bit "digits" along with descriptive
/  information.  The arrays of digits are only required to be
/  aligned on 32-bit boundary.  This implementation works only
/  when the two factors and the result happen to be 64 bit aligned
/  and have an even number of digits.
/
/ ---------------------------------------------------------------------------

/ r = a * digit, r and a are vectors of length len
/ returns the carry digit
/ r and a are 64 bit aligned.
/
/ uint64_t
/ big_mul_set_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
/ big_mul_set_vec64(uint64_t *r, uint64_t *a, int len, uint64_t digit)
/
        ENTRY(big_mul_set_vec)
        ENTRY(big_mul_set_vec64)
        xorq    %rax, %rax              / if (len == 0) return (0)
        testq   %rdx, %rdx
        jz      .L17

        movq    %rdx, %r8               / Use r8 for len; %rdx is used by mul
        xorq    %r9, %r9                / cy = 0

.L15:
        cmpq    $8, %r8                 / 8 - len
        jb      .L16
        movq    0(%rsi), %rax           / rax = a[0]
        movq    8(%rsi), %r11           / prefetch a[1]
        mulq    %rcx                    / p = a[0] * digit
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 0(%rdi)           / r[0] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    16(%rsi), %r11          / prefetch a[2]
        mulq    %rcx                    / p = a[1] * digit
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 8(%rdi)           / r[1] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    24(%rsi), %r11          / prefetch a[3]
        mulq    %rcx                    / p = a[2] * digit
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 16(%rdi)          / r[2] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    32(%rsi), %r11          / prefetch a[4]
        mulq    %rcx                    / p = a[3] * digit
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 24(%rdi)          / r[3] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)
```

```
110         movq    %r11, %rax
111         movq    40(%rsi), %r11          / prefetch a[5]
112         mulq    %rcx                    / p = a[4] * digit
113         addq    %r9, %rax
114         adcq    $0, %rdx                / p += cy
115         movq    %rax, 32(%rdi)          / r[4] = lo(p)
116         movq    %rdx, %r9               / cy = hi(p)

118         movq    %r11, %rax
119         movq    48(%rsi), %r11          / prefetch a[6]
120         mulq    %rcx                    / p = a[5] * digit
121         addq    %r9, %rax
122         adcq    $0, %rdx                / p += cy
123         movq    %rax, 40(%rdi)          / r[5] = lo(p)
124         movq    %rdx, %r9               / cy = hi(p)

126         movq    %r11, %rax
127         movq    56(%rsi), %r11          / prefetch a[7]
128         mulq    %rcx                    / p = a[6] * digit
129         addq    %r9, %rax
130         adcq    $0, %rdx                / p += cy
131         movq    %rax, 48(%rdi)          / r[6] = lo(p)
132         movq    %rdx, %r9               / cy = hi(p)

134         movq    %r11, %rax
135         mulq    %rcx                    / p = a[7] * digit
136         addq    %r9, %rax
137         adcq    $0, %rdx                / p += cy
138         movq    %rax, 56(%rdi)          / r[7] = lo(p)
139         movq    %rdx, %r9               / cy = hi(p)

141         addq    $64, %rsi
142         addq    $64, %rdi
143         subq    $8, %r8

145         jz      .L17
146         jmp     .L15

148 .L16:
149         movq    0(%rsi), %rax
150         mulq    %rcx                    / p = a[0] * digit
151         addq    %r9, %rax
152         adcq    $0, %rdx                / p += cy
153         movq    %rax, 0(%rdi)           / r[0] = lo(p)
154         movq    %rdx, %r9               / cy = hi(p)
155         decq    %r8
156         jz      .L17

158         movq    8(%rsi), %rax
159         mulq    %rcx                    / p = a[1] * digit
160         addq    %r9, %rax
161         adcq    $0, %rdx                / p += cy
162         movq    %rax, 8(%rdi)           / r[1] = lo(p)
163         movq    %rdx, %r9               / cy = hi(p)
164         decq    %r8
165         jz      .L17

167         movq    16(%rsi), %rax
168         mulq    %rcx                    / p = a[2] * digit
169         addq    %r9, %rax
170         adcq    $0, %rdx                / p += cy
171         movq    %rax, 16(%rdi)          / r[2] = lo(p)
172         movq    %rdx, %r9               / cy = hi(p)
173         decq    %r8
174         jz      .L17
```



```
176         movq    24(%rsi), %rax
177         mulq    %rcx                    / p = a[3] * digit
178         addq    %r9, %rax
179         adcq    $0, %rdx                / p += cy
180         movq    %rax, 24(%rdi)          / r[3] = lo(p)
181         movq    %rdx, %r9               / cy = hi(p)
182         decq    %r8
183         jz      .L17

185         movq    32(%rsi), %rax
186         mulq    %rcx                    / p = a[4] * digit
187         addq    %r9, %rax
188         adcq    $0, %rdx                / p += cy
189         movq    %rax, 32(%rdi)          / r[4] = lo(p)
190         movq    %rdx, %r9               / cy = hi(p)
191         decq    %r8
192         jz      .L17

194         movq    40(%rsi), %rax
195         mulq    %rcx                    / p = a[5] * digit
196         addq    %r9, %rax
197         adcq    $0, %rdx                / p += cy
198         movq    %rax, 40(%rdi)          / r[5] = lo(p)
199         movq    %rdx, %r9               / cy = hi(p)
200         decq    %r8
201         jz      .L17

203         movq    48(%rsi), %rax
204         mulq    %rcx                    / p = a[6] * digit
205         addq    %r9, %rax
206         adcq    $0, %rdx                / p += cy
207         movq    %rax, 48(%rdi)          / r[6] = lo(p)
208         movq    %rdx, %r9               / cy = hi(p)
209         decq    %r8
210         jz      .L17


213 .L17:
214         movq    %r9, %rax
215         ret
216         SET_SIZE(big_mul_set_vec)
222         SET_SIZE(big_mul_set_vec64)


219 / ------------------------------------------------------------------------
220 /
221 /  Implementation of big_mul_add_vec which exploits
222 /  the 64X64->128 bit  unsigned multiply instruction.
223 /
224 /  As defined in Sun's bignum library for pkcs11, bignums are
225 /  composed of an array of 64-bit "digits" or "chunks" along with
226 /  descriptive information.
230 /  composed of an array of 32-bit "digits" along with descriptive
231 /  information.  The arrays of digits are only required to be
232 /  aligned on 32-bit boundary.  This implementation works only
233 /  when the two factors and the result happen to be 64 bit aligned
234 /  and have an even number of digits.
227 /
228 / ------------------------------------------------------------------------

230 / r += a * digit, r and a are vectors of length len
231 / returns the carry digit
232 / r and a are 64 bit aligned.
233 /
234 / uint64_t
235 / big_mul_add_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
```

```
/ big_mul_add_vec64(uint64_t *r, uint64_t *a, int len, uint64_t digit)
/
        ENTRY(big_mul_add_vec)
        ENTRY(big_mul_add_vec64)
        xorq    %rax, %rax              / if (len == 0) return (0)
        testq   %rdx, %rdx
        jz      .L27

        movq    %rdx, %r8               / Use r8 for len; %rdx is used by mul
        xorq    %r9, %r9                / cy = 0

.L25:
        cmpq    $8, %r8                 / 8 - len
        jb      .L26
        movq    0(%rsi), %rax           / rax = a[0]
        movq    0(%rdi), %r10           / r10 = r[0]
        movq    8(%rsi), %r11           / prefetch a[1]
        mulq    %rcx                    / p = a[0] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[0]
        movq    8(%rdi), %r10           / prefetch r[1]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 0(%rdi)           / r[0] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    16(%rsi), %r11          / prefetch a[2]
        mulq    %rcx                    / p = a[1] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[1]
        movq    16(%rdi), %r10          / prefetch r[2]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 8(%rdi)           / r[1] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    24(%rsi), %r11          / prefetch a[3]
        mulq    %rcx                    / p = a[2] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[2]
        movq    24(%rdi), %r10          / prefetch r[3]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 16(%rdi)          / r[2] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    32(%rsi), %r11          / prefetch a[4]
        mulq    %rcx                    / p = a[3] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[3]
        movq    32(%rdi), %r10          / prefetch r[4]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 24(%rdi)          / r[3] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    40(%rsi), %r11          / prefetch a[5]
        mulq    %rcx                    / p = a[4] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[4]
        movq    40(%rdi), %r10          / prefetch r[5]
        addq    %r9, %rax
```



```
        adcq    $0, %rdx                / p += cy
        movq    %rax, 32(%rdi)          / r[4] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    48(%rsi), %r11          / prefetch a[6]
        mulq    %rcx                    / p = a[5] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[5]
        movq    48(%rdi), %r10          / prefetch r[6]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 40(%rdi)          / r[5] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        movq    56(%rsi), %r11          / prefetch a[7]
        mulq    %rcx                    / p = a[6] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[6]
        movq    56(%rdi), %r10          / prefetch r[7]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 48(%rdi)          / r[6] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        movq    %r11, %rax
        mulq    %rcx                    / p = a[7] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[7]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 56(%rdi)          / r[7] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)

        addq    $64, %rsi
        addq    $64, %rdi
        subq    $8, %r8

        jz      .L27
        jmp     .L25

.L26:
        movq    0(%rsi), %rax
        movq    0(%rdi), %r10
        mulq    %rcx                    / p = a[0] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[0]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 0(%rdi)           / r[0] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)
        decq    %r8
        jz      .L27

        movq    8(%rsi), %rax
        movq    8(%rdi), %r10
        mulq    %rcx                    / p = a[1] * digit
        addq    %r10, %rax
        adcq    $0, %rdx                / p += r[1]
        addq    %r9, %rax
        adcq    $0, %rdx                / p += cy
        movq    %rax, 8(%rdi)           / r[1] = lo(p)
        movq    %rdx, %r9               / cy = hi(p)
        decq    %r8
        jz      .L27
```

```
367         movq    16(%rsi), %rax
368         movq    16(%rdi), %r10
369         mulq    %rcx                    / p = a[2] * digit
370         addq    %r10, %rax
371         adcq    $0, %rdx                / p += r[2]
372         addq    %r9, %rax
373         adcq    $0, %rdx                / p += cy
374         movq    %rax, 16(%rdi)          / r[2] = lo(p)
375         movq    %rdx, %r9               / cy = hi(p)
376         decq    %r8
377         jz      .L27

379         movq    24(%rsi), %rax
380         movq    24(%rdi), %r10
381         mulq    %rcx                    / p = a[3] * digit
382         addq    %r10, %rax
383         adcq    $0, %rdx                / p += r[3]
384         addq    %r9, %rax
385         adcq    $0, %rdx                / p += cy
386         movq    %rax, 24(%rdi)          / r[3] = lo(p)
387         movq    %rdx, %r9               / cy = hi(p)
388         decq    %r8
389         jz      .L27

391         movq    32(%rsi), %rax
392         movq    32(%rdi), %r10
393         mulq    %rcx                    / p = a[4] * digit
394         addq    %r10, %rax
395         adcq    $0, %rdx                / p += r[4]
396         addq    %r9, %rax
397         adcq    $0, %rdx                / p += cy
398         movq    %rax, 32(%rdi)          / r[4] = lo(p)
399         movq    %rdx, %r9               / cy = hi(p)
400         decq    %r8
401         jz      .L27

403         movq    40(%rsi), %rax
404         movq    40(%rdi), %r10
405         mulq    %rcx                    / p = a[5] * digit
406         addq    %r10, %rax
407         adcq    $0, %rdx                / p += r[5]
408         addq    %r9, %rax
409         adcq    $0, %rdx                / p += cy
410         movq    %rax, 40(%rdi)          / r[5] = lo(p)
411         movq    %rdx, %r9               / cy = hi(p)
412         decq    %r8
413         jz      .L27

415         movq    48(%rsi), %rax
416         movq    48(%rdi), %r10
417         mulq    %rcx                    / p = a[6] * digit
418         addq    %r10, %rax
419         adcq    $0, %rdx                / p += r[6]
420         addq    %r9, %rax
421         adcq    $0, %rdx                / p += cy
422         movq    %rax, 48(%rdi)          / r[6] = lo(p)
423         movq    %rdx, %r9               / cy = hi(p)
424         decq    %r8
425         jz      .L27


428 .L27:
429         movq    %r9, %rax
430         ret
431         SET_SIZE(big_mul_add_vec)
```



```
439         SET_SIZE(big_mul_add_vec64)


434 / void
435 / big_sqr_vec(uint64_t *r, uint64_t *a, int len)
443 / big_sqr_vec64(uint64_t *r, uint64_t *a, int len)

437         ENTRY(big_sqr_vec)
445         ENTRY(big_sqr_vec64)
438         pushq   %rbx
439         pushq   %rbp
440         pushq   %r12
441         pushq   %r13
442         pushq   %r14
443         pushq   %r15
444         pushq   %rdx                    / save arg3, len
445         pushq   %rsi                    / save arg2, a
446         pushq   %rdi                    / save arg1, r

448         leaq    8(%rdi), %r13           / tr = r + 1
449         movq    %rsi, %r14              / ta = a
450         movq    %rdx, %r15              / tlen = len
451         decq    %r15                    / tlen = len - 1
452         movq    %r13, %rdi              / arg1 = tr
453         leaq    8(%r14), %rsi           / arg2 = ta + 1
454         movq    %r15, %rdx              / arg3 = tlen
455         movq    0(%r14), %rcx           / arg4 = ta[0]
456         call    big_mul_set_vec
464         call    big_mul_set_vec64
457         movq    %rax, 0(%r13, %r15, 8)  / tr[tlen] = cy
458 .L31:
459         decq    %r15                    / --tlen
460         jz      .L32                    / while (--tlen != 0)

462         addq    $16, %r13               / tr += 2
463         addq    $8, %r14                / ++ta
464         movq    %r13, %rdi              / arg1 = tr
465         leaq    8(%r14), %rsi           / arg2 = ta + 1
466         movq    %r15, %rdx              / arg3 = tlen
467         movq    0(%r14), %rcx           / arg4 = ta[0]
468         call    big_mul_add_vec
476         call    big_mul_add_vec64
469         movq    %rax, 0(%r13, %r15, 8)  / tr[tlen] = cy
470         jmp     .L31

472 .L32:

474 / No more function calls after this.
475 / Restore arguments to registers.
476 / However, don't use %rdx for arg3, len, because it is heavily
477 / used by the hardware MUL instruction.  Use %r8, instead.
478         movq    0(%rsp), %rdi           / %rdi == arg1 == r
479         movq    8(%rsp), %rsi           / %rsi == arg2 == a
480         movq    16(%rsp), %r8           / %r8  == arg3 == len

482         movq    0(%rsi), %rax           / %rax = a[0];
483         mulq    %rax                    / s = %edx:%eax = a[0]**2
484         movq    %rax, 0(%rdi)           / r[0] = lo64(s)
485         movq    %rdx, %r9               / cy = hi64(s)
486         xorq    %rdx, %rdx
487         movq    8(%rdi), %rax           / p = %rdx:%rax = r[1]
488         addq    %rax, %rax
489         adcq    $0, %rdx                / p = p << 1
490         addq    %r9, %rax
491         adcq    $0, %rdx                / p = (r[1] << 1) + cy
492         movq    %rax, 8(%rdi)           / r[1] = lo64(p)
```

```
493         movq    %rdx, %r9               / cy = hi64(p)
494         movq    $1, %r11                / row = 1
495         movq    $2, %r12                / col = 2
496         movq    %r8, %r15
497         decq    %r15                    / tlen = len - 1
498 .L33:
499         cmpq    %r8, %r11               / len - row
500         jae     .L34                    / while (row < len)

502         movq    0(%rsi, %r11, 8), %rax  / s = (uint128_t)a[row]
503         mulq    %rax                    / s = s * s
504         xorq    %rbx, %rbx
505         movq    0(%rdi, %r12, 8), %rcx  / p = (uint128_t)r[col]
506         addq    %rcx, %rcx
507         adcq    $0, %rbx                / p = p << 1
508         addq    %rcx, %rax
509         adcq    %rbx, %rdx              / t = p + s
510         xorq    %r10, %r10
511         movq    %rax, %rbp              / t2 = 0:lo64(t)
512         addq    %r9, %rbp
513         adcq    $0, %r10                / t2 = %r10:%rbp = lo64(t) + cy
514         movq    %rbp, 0(%rdi, %r12, 8)  / r[col] = lo64(t2)
515         xorq    %rcx, %rcx
516         movq    %rdx, %r9
517         addq    %r10, %r9
518         adcq    $0, %rcx                / cy = hi64(t) + hi64(t2)
519         cmpq    %r11, %r15
520         je      .L34                    / if (row == len - 1) break
521         xorq    %rdx, %rdx
522         movq    8(%rdi, %r12, 8), %rax
523         addq    %rax, %rax
524         adcq    $0, %rdx
525         addq    %r9, %rax
526         adcq    %rcx, %rdx              / p = (lo64(r[col+1]) << 1) + cy
527         movq    %rax, 8(%rdi, %r12, 8)  / r[col+1] = lo64(p)
528         movq    %rdx, %r9               / cy = hi64(p)

530         incq    %r11                    / ++row
531         addq    $2, %r12                / col += 2
532         jmp     .L33

534 .L34:
535         movq    %r9, 8(%rdi, %r12, 8)   / r[col+1] = lo64(cy)

537         addq    $24, %rsp               / skip %rdi, %rsi, %rdx
538         popq    %r15
539         popq    %r14
540         popq    %r13
541         popq    %r12
542         popq    %rbp
543         popq    %rbx

545         ret

547         SET_SIZE(big_sqr_vec)
555         SET_SIZE(big_sqr_vec64)

549 #endif  /* lint */
```

```
**********************************************************
    7544 Wed Feb 25 22:19:38 2009
new/usr/src/common/bignum/bignum.h
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _BIGNUM_H
#define _BIGNUM_H

#pragma ident   "%Z%%M% %I%     %E% SMI"

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/types.h>

#if defined(__sparcv9) || defined(__amd64) /* 64-bit chunk size */
#ifndef __sparcv9
#define BIGNUM_CHUNK_32
#else
#ifndef UMUL64
#define UMUL64  /* 64-bit multiplication results are supported */
#define UMUL64
#endif
#else
#define BIGNUM_CHUNK_32
#endif


#define BITSINBYTE      8

/* Bignum "digits" (aka "chunks" or "words") are either 32- or 64-bits */
#ifdef BIGNUM_CHUNK_32
#define BIG_CHUNK_SIZE          32
#define BIG_CHUNK_TYPE          uint32_t
#define BIG_CHUNK_TYPE_SIGNED   int32_t
#define BIG_CHUNK_HIGHBIT       0x80000000
#define BIG_CHUNK_ALLBITS       0xffffffff
```



```
#define BIG_CHUNK_LOWHALFBITS   0xffff
#define BIG_CHUNK_HALF_HIGHBIT  0x8000

#else
#define BIG_CHUNK_SIZE          64
#define BIG_CHUNK_TYPE          uint64_t
#define BIG_CHUNK_TYPE_SIGNED   int64_t
#define BIG_CHUNK_HIGHBIT       0x8000000000000000ULL
#define BIG_CHUNK_ALLBITS       0xffffffffffffffffULL
#define BIG_CHUNK_LOWHALFBITS   0xffffffffULL
#define BIG_CHUNK_HALF_HIGHBIT  0x80000000ULL
#endif

#define BITLEN2BIGNUMLEN(x)     (((x) + BIG_CHUNK_SIZE - 1) / BIG_CHUNK_SIZE)
#define CHARLEN2BIGNUMLEN(x)    (((x) + sizeof (BIG_CHUNK_TYPE) - 1) / \
                                    sizeof (BIG_CHUNK_TYPE))

#define BIGNUM_WORDSIZE (BIG_CHUNK_SIZE / BITSINBYTE)  /* word size in bytes */
#define BIG_CHUNKS_FOR_160BITS  ((160 + BIG_CHUNK_SIZE - 1) / BIG_CHUNK_SIZE)


/*
 * leading 0's are permitted
 * 0 should be represented by size>=1, size>=len>=1, sign=1,
 * value[i]=0 for 0<i<len
 */
typedef struct {
        /* size and len in units of BIG_CHUNK_TYPE words  */
        uint32_t        size;   /* size of memory allocated for value  */
        uint32_t        len;    /* number of valid data words in value */
        int size; /* size of memory allocated for value   */
        int len;  /* number of words that hold valid data in value */
        int             sign;   /* 1 for nonnegative, -1 for negative  */
        int             malloced; /* 1 if value was malloced, 0 if not */
        int malloced; /* 1 if value was malloced 0 if not */
        BIG_CHUNK_TYPE *value;
} BIGNUM;
______unchanged_portion_omitted_
```

```
**********************************************************
   64704 Wed Feb 25 22:19:50 2009
new/usr/src/common/bignum/bignumimpl.c
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#define big_div_pos_fast big_div_pos

#include "bignum.h"

/*
 * Configuration guide
 * -------------------
 *
 * There are 4 preprocessor symbols used to configure the bignum
 * implementation.  This file contains no logic to configure based on
 * processor; we leave that to the Makefiles to specify.
 *
 * USE_FLOATING_POINT
 *   Meaning: There is support for a fast floating-point implementation of
 *   Montgomery multiply.
 *
 * PSR_MUL
 *   Meaning: There are processor-specific versions of the low level
 *   functions to implement big_mul.  Those functions are: big_mul_set_vec,
 *   big_mul_add_vec, big_mul_vec, and big_sqr_vec.  PSR_MUL implies support
 *   for all 4 functions.  You cannot pick and choose which subset of these
 *   functions to support; that would lead to a rat's nest of #ifdefs.
 *
 * HWCAP
 *   Meaning: Call multiply support functions through a function pointer.
 *   On x86, there are multiple implementations for different hardware
 *   On x86, there are multiple implementations for differnt hardware
 *   capabilities, such as MMX, SSE2, etc.  Tests are made at run-time, when
 *   a function is first used.  So, the support functions are called through
 *   a function pointer.  There is no need for that on Sparc, because there
 *   is only one implementation; support functions are called directly.
```



```
 *   Later, if there were some new VIS instruction, or something, and a
 *   run-time test were needed, rather than variant kernel modules and
 *   libraries, then HWCAP would be defined for Sparc, as well.
 *
 * UMUL64
 *   Meaning: It is safe to use generic C code that assumes the existence
 *   of a 32 x 32 --> 64 bit unsigned multiply.  If this is not defined,
 *   then the generic code for big_mul_add_vec() must necessarily be very slow,
 *   because it must fall back to using 16 x 16 --> 32 bit multiplication.
 *
 */


#include <sys/types.h>
#include "bignum.h"

#ifdef  _KERNEL
#include <sys/ddi.h>
#include <sys/mdesc.h>
#include <sys/crypto/common.h>

#include <sys/types.h>
#include <sys/kmem.h>
#include <sys/param.h>
#include <sys/sunddi.h>

#else
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#define ASSERT  assert
#endif  /* _KERNEL */

#ifdef  _LP64 /* truncate 64-bit size_t to 32-bits */
#define UI32(ui)        ((uint32_t)ui)
#else /* size_t already 32-bits */
#define UI32(ui)        (ui)
#endif


#ifdef  _KERNEL
#define big_malloc(size)        kmem_alloc(size, KM_NOSLEEP)
#define big_free(ptr, size)     kmem_free(ptr, size)

void *
big_realloc(void *from, size_t oldsize, size_t newsize)
{
        void *rv;

        rv = kmem_alloc(newsize, KM_NOSLEEP);
        if (rv != NULL)
                bcopy(from, rv, oldsize);
        kmem_free(from, oldsize);
        return (rv);
}

#else   /* _KERNEL */

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#define ASSERT  assert

#ifndef MALLOC_DEBUG

#define big_malloc(size)        malloc(size)
```

```
112 #define big_free(ptr, size)     free(ptr)

114 #else

116 void
117 big_free(void *ptr, size_t size)
118 {
119         printf("freed %d bytes at %p\n", size, ptr);
120         free(ptr);
121 }
_____unchanged_portion_omitted_
131 #endif /* MALLOC_DEBUG */

133 #define big_realloc(x, y, z) realloc((x), (z))


136 /*
137  * printbignum()
138  * Print a BIGNUM type to stdout.
139  */
140 void
141 printbignum(char *aname, BIGNUM *a)
142 {
143         int i;

145         (void) printf("\n%s\n%d\n", aname, a->sign*a->len);
146         for (i = a->len - 1; i >= 0; i--) {
147 #ifdef BIGNUM_CHUNK_32
148                 (void) printf("%08x ", a->value[i]);
149                 if (((i & (BITSINBYTE - 1)) == 0) && (i != 0)) {
138                 if ((i % 8 == 0) && (i != 0)) {
150                         (void) printf("\n");
151                 }
152 #else
153                 (void) printf("%08x %08x ", (uint32_t)((a->value[i]) >> 32),
154                     (uint32_t)((a->value[i]) & 0xffffffff));
155                 if (((i & 3) == 0) && (i != 0)) { /* end of this chunk */
144                 if ((i % 4 == 0) && (i != 0)) {
156                         (void) printf("\n");
157                 }
158 #endif
159         }
160         (void) printf("\n");
161 }

163 #endif  /* _KERNEL */


166 /*
167  * big_init()
168  * Initialize and allocate memory for a BIGNUM type.
169  *
170  * big_init(number, size) is equivalent to big_init1(number, size, NULL, 0)
171  *
172  * Note: call big_finish() to free memory allocated by big_init().
173  *
174  * Input:
175  * number       Uninitialized memory for BIGNUM
176  * size         Minimum size, in BIG_CHUNK_SIZE-bit words, required for BIGNUM
177  *
178  * Output:
179  * number       Initialized BIGNUM
180  *
181  * Return BIG_OK on success or BIG_NO_MEM for an allocation error.
182  */
155 /* size in BIG_CHUNK_SIZE-bit words */
```



```
183 BIG_ERR_CODE
184 big_init(BIGNUM *number, int size)
185 {
186         number->value = big_malloc(BIGNUM_WORDSIZE * size);
159         number->value = big_malloc(sizeof (BIG_CHUNK_TYPE) * size);
187         if (number->value == NULL) {
188                 return (BIG_NO_MEM);
189         }
190         number->size = size;
191         number->len = 0;
192         number->sign = 1;
193         number->malloced = 1;
194         return (BIG_OK);
195 }


198 /*
199  * big_init1()
200  * Initialize and, if needed, allocate memory for a BIGNUM type.
201  * Use the buffer passed, buf, if any, instad of allocating memory
202  * if it's at least "size" bytes.
203  *
204  * Note: call big_finish() to free memory allocated by big_init().
205  *
206  * Input:
207  * number       Uninitialized memory for BIGNUM
208  * size         Minimum size, in BIG_CHUNK_SIZE-bit words, required for BIGNUM
209  * buf          Buffer for storing a BIGNUM.
210  *              If NULL, big_init1() will allocate a buffer
211  * bufsize      Size, in BIG_CHUNK_SIZE_bit words, of buf
212  *
213  * Output:
214  * number       Initialized BIGNUM
215  *
216  * Return BIG_OK on success or BIG_NO_MEM for an allocation error.
217  */
170 /* size in BIG_CHUNK_SIZE-bit words */
218 BIG_ERR_CODE
219 big_init1(BIGNUM *number, int size, BIG_CHUNK_TYPE *buf, int bufsize)
220 {
221         if ((buf == NULL) || (size > bufsize)) {
222                 number->value = big_malloc(BIGNUM_WORDSIZE * size);
175                 number->value = big_malloc(sizeof (BIG_CHUNK_TYPE) * size);
223                 if (number->value == NULL) {
224                         return (BIG_NO_MEM);
225                 }
226                 number->size = size;
227                 number->malloced = 1;
228         } else {
229                 number->value = buf;
230                 number->size = bufsize;
231                 number->malloced = 0;
232         }
233         number->len = 0;
234         number->sign = 1;

236         return (BIG_OK);
237 }


240 /*
241  * big_finish()
242  * Free memory, if any, allocated by big_init() or big_init1().
243  */
244 void
245 big_finish(BIGNUM *number)
```

```
246 {
247         if (number->malloced == 1) {
248                 big_free(number->value, BIGNUM_WORDSIZE * number->size);
196                 big_free(number->value,
197                     sizeof (BIG_CHUNK_TYPE) * number->size);
249                 number->malloced = 0;
250         }
251 }


254 /*
255  * bn->size should be at least
256  * (len + BIGNUM_WORDSIZE - 1) / BIGNUM_WORDSIZE bytes
205  * (len + sizeof (BIG_CHUNK_TYPE) - 1) / sizeof (BIG_CHUNK_TYPE) bytes
257  * converts from byte-big-endian format to bignum format (words in
258  * little endian order, but bytes within the words big endian)
259  */
260 void
261 bytestring2bignum(BIGNUM *bn, uchar_t *kn, size_t len)
262 {
263         int             i, j;
264         uint32_t        offs;
265         const uint32_t  slen = UI32(len);
212         int             i, j, offs;
266         BIG_CHUNK_TYPE  word;
267         uchar_t         *knwordp;

269         if (slen == 0) {
270                 bn->len = 1;
271                 bn->value[0] = 0;
272                 return;
273         }
216 #ifdef  _LP64
217         offs = (uint32_t)len % sizeof (BIG_CHUNK_TYPE);
218         bn->len = (uint32_t)len / sizeof (BIG_CHUNK_TYPE);

275         offs = slen % BIGNUM_WORDSIZE;
276         bn->len = slen / BIGNUM_WORDSIZE;

278         for (i = 0; i < slen / BIGNUM_WORDSIZE; i++) {
279                 knwordp = &(kn[slen - BIGNUM_WORDSIZE * (i + 1)]);
220         for (i = 0; i < (uint32_t)len / sizeof (BIG_CHUNK_TYPE); i++) {
221 #else   /* !_LP64 */
222         offs = len % sizeof (BIG_CHUNK_TYPE);
223         bn->len = len / sizeof (BIG_CHUNK_TYPE);
224         for (i = 0; i < len / sizeof (BIG_CHUNK_TYPE); i++) {
225 #endif  /* _LP64 */
226                 knwordp = &(kn[len - sizeof (BIG_CHUNK_TYPE) * (i + 1)]);
280                 word = knwordp[0];
281                 for (j = 1; j < BIGNUM_WORDSIZE; j++) {
282                         word = (word << BITSINBYTE) + knwordp[j];
228                 for (j = 1; j < sizeof (BIG_CHUNK_TYPE); j++) {
229                         word = (word << 8)+ knwordp[j];
283                 }
284                 bn->value[i] = word;
285         }
286         if (offs > 0) {
287                 word = kn[0];
288                 for (i = 1; i < offs; i++) word = (word << BITSINBYTE) + kn[i];
235                 for (i = 1; i < offs; i++) word = (word << 8) + kn[i];
289                 bn->value[bn->len++] = word;
290         }
291         while ((bn->len > 1) && (bn->value[bn->len - 1] == 0)) {
238         while ((bn->len > 1) && (bn->value[bn->len-1] == 0)) {
292                 bn->len --;
293         }
```



```
294 }


297 /*
298  * copies the least significant len bytes if
299  * len < bn->len * BIGNUM_WORDSIZE
245  * len < bn->len * sizeof (BIG_CHUNK_TYPE)
300  * converts from bignum format to byte-big-endian format.
301  * bignum format is words of type  BIG_CHUNK_TYPE in little endian order.
302  */
303 void
304 bignum2bytestring(uchar_t *kn, BIGNUM *bn, size_t len)
305 {
306         int             i, j;
307         uint32_t        offs;
308         const uint32_t  slen = UI32(len);
252         int             i, j, offs;
309         BIG_CHUNK_TYPE  word;

311         if (len < BIGNUM_WORDSIZE * bn->len) {
312                 for (i = 0; i < slen / BIGNUM_WORDSIZE; i++) {
255         if (len < sizeof (BIG_CHUNK_TYPE) * bn->len) {
256 #ifdef  _LP64
257                 for (i = 0; i < (uint32_t)len / sizeof (BIG_CHUNK_TYPE); i++) {
258 #else   /* !_LP64 */
259                 for (i = 0; i < len / sizeof (BIG_CHUNK_TYPE); i++) {
260 #endif  /* _LP64 */
313                         word = bn->value[i];
314                         for (j = 0; j < BIGNUM_WORDSIZE; j++) {
315                                 kn[slen - BIGNUM_WORDSIZE * i - j - 1] =
262                         for (j = 0; j < sizeof (BIG_CHUNK_TYPE); j++) {
263                                 kn[len - sizeof (BIG_CHUNK_TYPE) * i - j - 1] =
316                                     word & 0xff;
317                                 word = word >> BITSINBYTE;
265                                 word = word >> 8;
318                         }
319                 }
320                 offs = slen % BIGNUM_WORDSIZE;
268 #ifdef  _LP64
269                 offs = (uint32_t)len % sizeof (BIG_CHUNK_TYPE);
270 #else   /* !_LP64 */
271                 offs = len % sizeof (BIG_CHUNK_TYPE);
272 #endif  /* _LP64 */
321                 if (offs > 0) {
322                         word = bn->value[slen / BIGNUM_WORDSIZE];
323                         for (i =  slen % BIGNUM_WORDSIZE; i > 0; i --) {
274                         word = bn->value[len / sizeof (BIG_CHUNK_TYPE)];
275 #ifdef  _LP64
276                         for (i =  (uint32_t)len % sizeof (BIG_CHUNK_TYPE);
277                             i > 0; i --) {
278 #else   /* !_LP64 */
279                         for (i = len % sizeof (BIG_CHUNK_TYPE);
280                             i > 0; i --) {
281 #endif  /* _LP64 */
324                                 kn[i - 1] = word & 0xff;
325                                 word = word >> BITSINBYTE;
283                                 word = word >> 8;
326                         }
327                 }
328         } else {
329                 for (i = 0; i < bn->len; i++) {
330                         word = bn->value[i];
331                         for (j = 0; j < BIGNUM_WORDSIZE; j++) {
332                                 kn[slen - BIGNUM_WORDSIZE * i - j - 1] =
289                         for (j = 0; j < sizeof (BIG_CHUNK_TYPE); j++) {
290                                 kn[len - sizeof (BIG_CHUNK_TYPE) * i - j - 1] =
```

```
                                    word & 0xff;
                                word = word >> BITSINBYTE;
                                word = word >> 8;
                        }
                }
                for (i = 0; i < slen - BIGNUM_WORDSIZE * bn->len; i++) {
#ifdef  _LP64
                for (i = 0;
                    i < (uint32_t)len - sizeof (BIG_CHUNK_TYPE) * bn->len;
                    i++) {
#else   /* !_LP64 */
                for (i = 0; i < len - sizeof (BIG_CHUNK_TYPE) * bn->len; i++) {
#endif  /* _LP64 */
                        kn[i] = 0;
                }
        }
}


int
big_bitlength(BIGNUM *a)
{
        int             l = 0, b = 0;
        BIG_CHUNK_TYPE  c;

        l = a->len - 1;
        while ((l > 0) && (a->value[l] == 0)) {
                l--;
        }
        b = BIG_CHUNK_SIZE;
        b = sizeof (BIG_CHUNK_TYPE) * BITSINBYTE;
        c = a->value[l];
        while ((b > 1) && ((c & BIG_CHUNK_HIGHBIT) == 0)) {
                c = c << 1;
                b--;
        }

        return (l * BIG_CHUNK_SIZE + b);
        return (l * sizeof (BIG_CHUNK_TYPE) * BITSINBYTE + b);
}


BIG_ERR_CODE
big_copy(BIGNUM *dest, BIGNUM *src)
{
        BIG_CHUNK_TYPE  *newptr;
        int             i, len;

        len = src->len;
        while ((len > 1) && (src->value[len - 1] == 0)) {
                len--;
        }
        src->len = len;
        if (dest->size < len) {
                if (dest->malloced == 1) {
                        newptr = (BIG_CHUNK_TYPE *)big_realloc(dest->value,
                            BIGNUM_WORDSIZE * dest->size,
                            BIGNUM_WORDSIZE * len);
                            sizeof (BIG_CHUNK_TYPE) * dest->size,
                            sizeof (BIG_CHUNK_TYPE) * len);
                } else {
                        newptr = (BIG_CHUNK_TYPE *)
                            big_malloc(BIGNUM_WORDSIZE * len);
                            big_malloc(sizeof (BIG_CHUNK_TYPE) * len);
                        if (newptr != NULL) {
                                dest->malloced = 1;
```



```
                        }
                }
                if (newptr == NULL) {
                        return (BIG_NO_MEM);
                }
                dest->value = newptr;
                dest->size = len;
        }
        dest->len = len;
        dest->sign = src->sign;
        for (i = 0; i < len; i++) {
                dest->value[i] = src->value[i];
        }

        return (BIG_OK);
}


BIG_ERR_CODE
big_extend(BIGNUM *number, int size)
{
        BIG_CHUNK_TYPE  *newptr;
        int             i;

        if (number->size >= size)
                return (BIG_OK);
        if (number->malloced) {
                number->value = big_realloc(number->value,
                    BIGNUM_WORDSIZE * number->size,
                    BIGNUM_WORDSIZE * size);
                    sizeof (BIG_CHUNK_TYPE) * number->size,
                    sizeof (BIG_CHUNK_TYPE) * size);
        } else {
                newptr = big_malloc(BIGNUM_WORDSIZE * size);
                newptr = big_malloc(sizeof (BIG_CHUNK_TYPE) * size);
                if (newptr != NULL) {
                        for (i = 0; i < number->size; i++) {
                                newptr[i] = number->value[i];
                        }
                }
                number->value = newptr;
        }

        if (number->value == NULL) {
                return (BIG_NO_MEM);
        }

        number->size = size;
        number->malloced = 1;
        return (BIG_OK);
}
______unchanged_portion_omitted_


/* returns -1 if |aa|<|bb|, 0 if |aa|==|bb| 1 if |aa|>|bb| */
int
big_cmp_abs(BIGNUM *aa, BIGNUM *bb)
{
        int     i;

        if (aa->len > bb->len) {
                for (i = aa->len - 1; i > bb->len - 1; i--) {
                        if (aa->value[i] > 0) {
                                return (1);
                        }
                }
```

```
593         } else if (aa->len < bb->len) {
594                 for (i = bb->len - 1; i > aa->len - 1; i--) {
595                         if (bb->value[i] > 0) {
596                                 return (-1);
597                         }
598                 }
599         } else {
600                 i = aa->len - 1;
564                 i = aa->len-1;
601         }
602         for (; i >= 0; i--) {
603                 if (aa->value[i] > bb->value[i]) {
604                         return (1);
605                 } else if (aa->value[i] < bb->value[i]) {
606                         return (-1);
607                 }
608         }

610         return (0);
611 }
```

*unchanged_portion_omitted_*

```
935 /* it is assumed that result->size is big enough */
936 void
937 big_shiftright(BIGNUM *result, BIGNUM *aa, int offs)
938 {
939         int              i;
940         BIG_CHUNK_TYPE  cy, ai;

942         if (offs == 0) {
943                 if (result != aa) {
944                         (void) big_copy(result, aa);
945                 }
946                 return;
947         }
948         cy = aa->value[0] >> offs;
949         for (i = 1; i < aa->len; i++) {
950                 ai = aa->value[i];
951                 result->value[i - 1] = (ai << (BIG_CHUNK_SIZE - offs)) | cy;
915                 result->value[i-1] = (ai << (BIG_CHUNK_SIZE - offs)) | cy;
952                 cy = ai >> offs;
953         }
954         result->len = aa->len;
955         result->value[result->len - 1] = cy;
956         result->sign = aa->sign;
957 }


960 /*
961  * result = aa/bb   remainder = aa mod bb
962  * it is assumed that aa and bb are positive
963  */
964 BIG_ERR_CODE
965 big_div_pos(BIGNUM *result, BIGNUM *remainder, BIGNUM *aa, BIGNUM *bb)
929 big_div_pos_fast(BIGNUM *result, BIGNUM *remainder, BIGNUM *aa, BIGNUM *bb)
966 {
967         BIG_ERR_CODE    err = BIG_OK;
968         int             i, alen, blen, tlen, rlen, offs;
969         BIG_CHUNK_TYPE  higha, highb, coeff;
970         BIG_CHUNK_TYPE  *a, *b;
971         BIGNUM          bbhigh, bblow, tresult, tmp1, tmp2;
972         BIG_CHUNK_TYPE  tmp1value[BIGTMPSIZE];
973         BIG_CHUNK_TYPE  tmp2value[BIGTMPSIZE];
974         BIG_CHUNK_TYPE  tresultvalue[BIGTMPSIZE];
975         BIG_CHUNK_TYPE  bblowvalue[BIGTMPSIZE];
```



```
976         BIG_CHUNK_TYPE  bbhighvalue[BIGTMPSIZE];

978         a = aa->value;
979         b = bb->value;
980         alen = aa->len;
981         blen = bb->len;
982         while ((alen > 1) && (a[alen - 1] == 0)) {
983                 alen = alen - 1;
984         }
985         aa->len = alen;
986         while ((blen > 1) && (b[blen - 1] == 0)) {
987                 blen = blen - 1;
988         }
989         bb->len = blen;
990         if ((blen == 1) && (b[0] == 0)) {
991                 return (BIG_DIV_BY_0);
992         }

994         if (big_cmp_abs(aa, bb) < 0) {
995                 if ((remainder != NULL) &&
996                     ((err = big_copy(remainder, aa)) != BIG_OK)) {
997                         return (err);
998                 }
999                 if (result != NULL) {
1000                        result->len = 1;
1001                        result->sign = 1;
1002                        result->value[0] = 0;
1003                }
1004                return (BIG_OK);
1005        }

1007        if ((err = big_init1(&bblow, blen + 1,
1008            bblowvalue, arraysize(bblowvalue))) != BIG_OK)
1009                return (err);

1011        if ((err = big_init1(&bbhigh, blen + 1,
1012            bbhighvalue, arraysize(bbhighvalue))) != BIG_OK)
1013                goto ret1;

1015        if ((err = big_init1(&tmp1, alen + 2,
1016            tmp1value, arraysize(tmp1value))) != BIG_OK)
1017                goto ret2;

1019        if ((err = big_init1(&tmp2, blen + 2,
1020            tmp2value, arraysize(tmp2value))) != BIG_OK)
1021                goto ret3;

1023        if ((err = big_init1(&tresult, alen - blen + 2,
1024            tresultvalue, arraysize(tresultvalue))) != BIG_OK)
1025                goto ret4;

1027        offs = 0;
1028        highb = b[blen - 1];
1029        if (highb >= (BIG_CHUNK_HALF_HIGHBIT << 1)) {
1030                highb = highb >> (BIG_CHUNK_SIZE / 2);
1031                offs = (BIG_CHUNK_SIZE / 2);
1032        }
1033        while ((highb & BIG_CHUNK_HALF_HIGHBIT) == 0) {
1034                highb = highb << 1;
1035                offs++;
1036        }

1038        big_shiftleft(&bblow, bb, offs);

1040        if (offs <= (BIG_CHUNK_SIZE / 2 - 1)) {
1041                big_shiftleft(&bbhigh, &bblow, BIG_CHUNK_SIZE / 2);
```

```
1042         } else {
1043                 big_shiftright(&bbhigh, &bblow, BIG_CHUNK_SIZE / 2);
1044         }
1045         if (bbhigh.value[bbhigh.len - 1] == 0) {
1046                 bbhigh.len--;
1047         } else {
1048                 bbhigh.value[bbhigh.len] = 0;
1049         }

1051         highb = bblow.value[bblow.len - 1];

1053         big_shiftleft(&tmp1, aa, offs);
1054         rlen = tmp1.len - bblow.len + 1;
1055         tresult.len = rlen;

1057         tmp1.len++;
1058         tlen = tmp1.len;
1059         tmp1.value[tmp1.len - 1] = 0;
1060         for (i = 0; i < rlen; i++) {
1061                 higha = (tmp1.value[tlen - 1] << (BIG_CHUNK_SIZE / 2)) +
1062                     (tmp1.value[tlen - 2] >> (BIG_CHUNK_SIZE / 2));
1063                 coeff = higha / (highb + 1);
1064                 big_mulhalf_high(&tmp2, &bblow, coeff);
1065                 big_sub_pos_high(&tmp1, &tmp1, &tmp2);
1066                 bbhigh.len++;
1067                 while (tmp1.value[tlen - 1] > 0) {
1068                         big_sub_pos_high(&tmp1, &tmp1, &bbhigh);
1069                         coeff++;
1070                 }
1071                 bbhigh.len--;
1072                 tlen--;
1073                 tmp1.len--;
1074                 while (big_cmp_abs_high(&tmp1, &bbhigh) >= 0) {
1075                         big_sub_pos_high(&tmp1, &tmp1, &bbhigh);
1076                         coeff++;
1077                 }
1078                 tresult.value[rlen - i - 1] = coeff << (BIG_CHUNK_SIZE / 2);
1079                 higha = tmp1.value[tlen - 1];
1080                 coeff = higha / (highb + 1);
1081                 big_mulhalf_low(&tmp2, &bblow, coeff);
1082                 tmp2.len--;
1083                 big_sub_pos_high(&tmp1, &tmp1, &tmp2);
1084                 while (big_cmp_abs_high(&tmp1, &bblow) >= 0) {
1085                         big_sub_pos_high(&tmp1, &tmp1, &bblow);
1086                         coeff++;
1087                 }
1088                 tresult.value[rlen - i - 1] =
1089                     tresult.value[rlen - i - 1] + coeff;
1090         }

1092         big_shiftright(&tmp1, &tmp1, offs);

1094         err = BIG_OK;

1096         if ((remainder != NULL) &&
1097             ((err = big_copy(remainder, &tmp1)) != BIG_OK))
1098                 goto ret;

1100         if (result != NULL)
1101                 err = big_copy(result, &tresult);

1103 ret:
1104         big_finish(&tresult);
1105 ret4:
1106         big_finish(&tmp1);
1107 ret3:
```



```
1108         big_finish(&tmp2);
1109 ret2:
1110         big_finish(&bbhigh);
1111 ret1:
1112         big_finish(&bblow);
1113         return (err);
1114 }


1117 /*
1118  * If there is no processor-specific integer implementation of
1119  * the lower level multiply functions, then this code is provided
1120  * for big_mul_set_vec(), big_mul_add_vec(), big_mul_vec() and
1121  * big_sqr_vec().
1122  *
1123  * There are two generic implementations.  One that assumes that
1124  * there is hardware and C compiler support for a 32 x 32 --> 64
1125  * bit unsigned multiply, but otherwise is not specific to any
1126  * processor, platform, or ISA.
1127  *
1128  * The other makes very few assumptions about hardware capabilities.
1129  * It does not even assume that there is any implementation of a
1130  * 32 x 32 --> 64 bit multiply that is accessible to C code and
1131  * appropriate to use.  It falls constructs 32 x 32 --> 64 bit
1132  * multiplies from 16 x 16 --> 32 bit multiplies.
1133  *
1134  */

1136 #if !defined(PSR_MUL)

1138 #ifdef UMUL64

1140 #if (BIG_CHUNK_SIZE == 32)

1142 #define UNROLL8

1144 #define MUL_SET_VEC_ROUND_PREFETCH(R) \
1145         p = pf * d; \
1146         pf = (uint64_t)a[R + 1]; \
1109         pf = (uint64_t)a[R+1]; \
1147         t = p + cy; \
1148         r[R] = (uint32_t)t; \
1149         cy = t >> 32

1151 #define MUL_SET_VEC_ROUND_NOPREFETCH(R) \
1152         p = pf * d; \
1153         t = p + cy; \
1154         r[R] = (uint32_t)t; \
1155         cy = t >> 32

1157 #define MUL_ADD_VEC_ROUND_PREFETCH(R) \
1158         t = (uint64_t)r[R]; \
1159         p = pf * d; \
1160         pf = (uint64_t)a[R + 1]; \
1123         pf = (uint64_t)a[R+1]; \
1161         t = p + t + cy; \
1162         r[R] = (uint32_t)t; \
1163         cy = t >> 32

1165 #define MUL_ADD_VEC_ROUND_NOPREFETCH(R) \
1166         t = (uint64_t)r[R]; \
1167         p = pf * d; \
1168         t = p + t + cy; \
1169         r[R] = (uint32_t)t; \
1170         cy = t >> 32
```

```
1172 #ifdef UNROLL8

1174 #define UNROLL 8

1176 /*
1177  * r = a * b
1178  * where r and a are vectors; b is a single 32-bit digit
1179  */

1181 uint32_t
1182 big_mul_set_vec(uint32_t *r, uint32_t *a, int len, uint32_t b)
1183 {
1184         uint64_t d, pf, p, t, cy;

1186         if (len == 0)
1187                 return (0);
1188         cy = 0;
1189         d = (uint64_t)b;
1190         pf = (uint64_t)a[0];
1191         while (len > UNROLL) {
1192                 MUL_SET_VEC_ROUND_PREFETCH(0);
1193                 MUL_SET_VEC_ROUND_PREFETCH(1);
1194                 MUL_SET_VEC_ROUND_PREFETCH(2);
1195                 MUL_SET_VEC_ROUND_PREFETCH(3);
1196                 MUL_SET_VEC_ROUND_PREFETCH(4);
1197                 MUL_SET_VEC_ROUND_PREFETCH(5);
1198                 MUL_SET_VEC_ROUND_PREFETCH(6);
1199                 MUL_SET_VEC_ROUND_PREFETCH(7);
1200                 r += UNROLL;
1201                 a += UNROLL;
1202                 len -= UNROLL;
1203         }
1204         if (len == UNROLL) {
1205                 MUL_SET_VEC_ROUND_PREFETCH(0);
1206                 MUL_SET_VEC_ROUND_PREFETCH(1);
1207                 MUL_SET_VEC_ROUND_PREFETCH(2);
1208                 MUL_SET_VEC_ROUND_PREFETCH(3);
1209                 MUL_SET_VEC_ROUND_PREFETCH(4);
1210                 MUL_SET_VEC_ROUND_PREFETCH(5);
1211                 MUL_SET_VEC_ROUND_PREFETCH(6);
1212                 MUL_SET_VEC_ROUND_NOPREFETCH(7);
1213                 return ((uint32_t)cy);
1214         }
1215         while (len > 1) {
1216                 MUL_SET_VEC_ROUND_PREFETCH(0);
1217                 ++r;
1218                 ++a;
1219                 --len;
1220         }
1221         if (len > 0) {
1222                 MUL_SET_VEC_ROUND_NOPREFETCH(0);
1223         }
1224         return ((uint32_t)cy);
1225 }
______unchanged_portion_omitted_
1277 #endif /* UNROLL8 */

1279 void
1280 big_sqr_vec(uint32_t *r, uint32_t *a, int len)
1281 {
1282         uint32_t        *tr, *ta;
1283         int             tlen, row, col;
1284         uint64_t        p, s, t, t2, cy;
1285         uint32_t        d;

1287         tr = r + 1;
```



```
1288         ta = a;
1289         tlen = len - 1;
1290         tr[tlen] = big_mul_set_vec(tr, ta + 1, tlen, ta[0]);
1291         while (--tlen > 0) {
1292                 tr += 2;
1293                 ++ta;
1294                 tr[tlen] = big_mul_add_vec(tr, ta + 1, tlen, ta[0]);
1295         }
1296         s = (uint64_t)a[0];
1297         s = s * s;
1298         r[0] = (uint32_t)s;
1299         cy = s >> 32;
1300         p = ((uint64_t)r[1] << 1) + cy;
1301         r[1] = (uint32_t)p;
1302         cy = p >> 32;
1303         row = 1;
1304         col = 2;
1305         while (row < len) {
1306                 s = (uint64_t)a[row];
1307                 s = s * s;
1308                 p = (uint64_t)r[col] << 1;
1309                 t = p + s;
1310                 d = (uint32_t)t;
1311                 t2 = (uint64_t)d + cy;
1312                 r[col] = (uint32_t)t2;
1313                 cy = (t >> 32) + (t2 >> 32);
1314                 if (row == len - 1)
1315                         break;
1316                 p = ((uint64_t)r[col + 1] << 1) + cy;
1317                 r[col + 1] = (uint32_t)p;
1279                 p = ((uint64_t)r[col+1] << 1) + cy;
1280                 r[col+1] = (uint32_t)p;
1318                 cy = p >> 32;
1319                 ++row;
1320                 col += 2;
1321         }
1322         r[col + 1] = (uint32_t)cy;
1285         r[col+1] = (uint32_t)cy;
1323 }
______unchanged_portion_omitted_

1393 void
1394 big_sqr_vec(BIG_CHUNK_TYPE *r, BIG_CHUNK_TYPE *a, int len)
1395 {
1396         int i;

1398         ASSERT(r != a);
1399         r[len] = big_mul_set_vec(r, a, len, a[0]);
1400         for (i = 1; i < len; ++i)
1401                 r[len + i] = big_mul_add_vec(r + i, a, len, a[i]);
1364                 r[len + i] = big_mul_add_vec(r+i, a, len, a[i]);
1402 }

1404 #endif /* BIG_CHUNK_SIZE == 32/64 */


1407 #else /* ! UMUL64 */

1409 #if (BIG_CHUNK_SIZE != 32)
1410 #error Don't use 64-bit chunks without defining UMUL64
1411 #endif


1414 /*
1415  * r = r + a * digit, r and a are vectors of length len
1416  * returns the carry digit
```

```
 */
uint32_t
big_mul_add_vec(uint32_t *r, uint32_t *a, int len, uint32_t digit)
{
        uint32_t cy, cy1, retcy, dlow, dhigh;
        int i;

        cy1 = 0;
        dlow = digit & 0xffff;
        dhigh = digit >> 16;
        for (i = 0; i < len; i++) {
                cy = (cy1 >> 16) + dlow * (a[i] & 0xffff) + (r[i] & 0xffff);
                cy1 = (cy >> 16) + dlow * (a[i]>>16) + (r[i] >> 16);
                r[i] = (cy & 0xffff) | (cy1 << 16);
        }
        retcy = cy1 >> 16;

        cy1 = r[0] & 0xffff;
        for (i = 0; i < len - 1; i++) {
                cy = (cy1 >> 16) + dhigh * (a[i] & 0xffff) + (r[i] >> 16);
                r[i] = (cy1 & 0xffff) | (cy << 16);
                cy1 = (cy >> 16) + dhigh * (a[i] >> 16) + (r[i + 1] & 0xffff);
        }
        cy = (cy1 >> 16) + dhigh * (a[len - 1] & 0xffff) + (r[len - 1] >> 16);
        r[len - 1] = (cy1 & 0xffff) | (cy << 16);
        retcy = (cy >> 16) + dhigh * (a[len - 1] >> 16) + retcy;

        return (retcy);
}
```

unchanged_portion_omitted_

```
void
big_sqr_vec(uint32_t *r, uint32_t *a, int len)
{
        int i;

        ASSERT(r != a);
        r[len] = big_mul_set_vec(r, a, len, a[0]);
        for (i = 1; i < len; ++i)
                r[len + i] = big_mul_add_vec(r + i, a, len, a[i]);
                r[len + i] = big_mul_add_vec(r+i, a, len, a[i]);
}

#endif /* UMUL64 */

void
big_mul_vec(BIG_CHUNK_TYPE *r, BIG_CHUNK_TYPE *a, int alen,
    BIG_CHUNK_TYPE *b, int blen)
{
        int i;

        r[alen] = big_mul_set_vec(r, a, alen, b[0]);
        for (i = 1; i < blen; ++i)
                r[alen + i] = big_mul_add_vec(r + i, a, alen, b[i]);
                r[alen + i] = big_mul_add_vec(r+i, a, alen, b[i]);
}


#endif /* ! PSR_MUL */


/*
 * result = aa * bb  result->value should be big enough to hold the result
 *
 * Implementation: Standard grammar school algorithm
 *
```



```
 */
BIG_ERR_CODE
big_mul(BIGNUM *result, BIGNUM *aa, BIGNUM *bb)
{
        BIGNUM          tmp1;
        BIG_CHUNK_TYPE  tmp1value[BIGTMPSIZE];
        BIG_CHUNK_TYPE  *r, *t, *a, *b;
        BIG_ERR_CODE    err;
        int             i, alen, blen, rsize, sign, diff;

        if (aa == bb) {
                diff = 0;
        } else {
                diff = big_cmp_abs(aa, bb);
                if (diff < 0) {
                        BIGNUM *tt;
                        tt = aa;
                        aa = bb;
                        bb = tt;
                }
        }
        a = aa->value;
        b = bb->value;
        alen = aa->len;
        blen = bb->len;
        while ((alen > 1) && (a[alen - 1] == 0)) {
                alen--;
        }
        aa->len = alen;
        while ((blen > 1) && (b[blen - 1] == 0)) {
                blen--;
        }
        bb->len = blen;

        rsize = alen + blen;
        ASSERT(rsize > 0);
        if (result->size < rsize) {
                err = big_extend(result, rsize);
                if (err != BIG_OK) {
                        return (err);
                }
                /* aa or bb might be an alias to result */
                a = aa->value;
                b = bb->value;
        }
        r = result->value;

        if (((alen == 1) && (a[0] == 0)) || ((blen == 1) && (b[0] == 0))) {
                result->len = 1;
                result->sign = 1;
                r[0] = 0;
                return (BIG_OK);
        }
        sign = aa->sign * bb->sign;
        if ((alen == 1) && (a[0] == 1)) {
                for (i = 0; i < blen; i++) {
                        r[i] = b[i];
                }
                result->len = blen;
                result->sign = sign;
                return (BIG_OK);
        }
        if ((blen == 1) && (b[0] == 1)) {
                for (i = 0; i < alen; i++) {
                        r[i] = a[i];
                }
```

```
1564                 result->len = alen;
1565                 result->sign = sign;
1566                 return (BIG_OK);
1567         }

1569         if ((err = big_init1(&tmp1, rsize,
1570             tmp1value, arraysize(tmp1value))) != BIG_OK) {
1571                 return (err);
1572         }
1573         (void) big_copy(&tmp1, aa);
1574         t = tmp1.value;

1576         for (i = 0; i < rsize; i++) {
1577                 t[i] = 0;
1578         }

1580         if (diff == 0 && alen > 2) {
1581                 BIG_SQR_VEC(t, a, alen);
1582         } else if (blen > 0) {
1583                 BIG_MUL_VEC(t, a, alen, b, blen);
1584         }

1586         if (t[rsize - 1] == 0) {
1587                 tmp1.len = rsize - 1;
1588         } else {
1589                 tmp1.len = rsize;
1590         }

1592         err = big_copy(result, &tmp1);

1552         if ((err = big_copy(result, &tmp1)) != BIG_OK) {
1553                 return (err);
1554         }
1594         result->sign = sign;

1596         big_finish(&tmp1);

1598         return (err);
1559         return (BIG_OK);
1599 }


1602 /*
1603  * caller must ensure that  a < n,  b < n  and  ret->size >=  2 * n->len + 1
1604  * and that ret is not n
1605  */
1606 BIG_ERR_CODE
1607 big_mont_mul(BIGNUM *ret, BIGNUM *a, BIGNUM *b, BIGNUM *n, BIG_CHUNK_TYPE n0)
1608 {
1609         int     i, j, nlen, needsubtract;
1610         BIG_CHUNK_TYPE  *nn, *rr;
1611         BIG_CHUNK_TYPE  digit, c;
1612         BIG_ERR_CODE    err;

1614         nlen = n->len;
1615         nn = n->value;

1617         rr = ret->value;

1619         if ((err = big_mul(ret, a, b)) != BIG_OK) {
1620                 return (err);
1621         }

1623         rr = ret->value;
1624         for (i = ret->len; i < 2 * nlen + 1; i++) {
1625                 rr[i] = 0;
```



```
1626         }
1627         for (i = 0; i < nlen; i++) {
1628                 digit = rr[i];
1629                 digit = digit * n0;

1631                 c = BIG_MUL_ADD_VEC(rr + i, nn, nlen, digit);
1632                 j = i + nlen;
1633                 rr[j] += c;
1634                 while (rr[j] < c) {
1635                         rr[j + 1] += 1;
1636                         j++;
1637                         c = 1;
1638                 }
1639         }

1641         needsubtract = 0;
1642         if ((rr[2 * nlen]  != 0))
1643                 needsubtract = 1;
1644         else {
1645                 for (i = 2 * nlen - 1; i >= nlen; i--) {
1646                         if (rr[i] > nn[i - nlen]) {
1647                                 needsubtract = 1;
1648                                 break;
1649                         } else if (rr[i] < nn[i - nlen]) {
1650                                 break;
1651                         }
1652                 }
1653         }
1654         if (needsubtract)
1655                 big_sub_vec(rr, rr + nlen, nn, nlen);
1656         else {
1657                 for (i = 0; i < nlen; i++) {
1658                         rr[i] = rr[i + nlen];
1659                 }
1660         }

1662         /* Remove leading zeros, but keep at least 1 digit: */
1663         for (i = nlen - 1; (i > 0) && (rr[i] == 0); i--)
1622         for (i = nlen - 1; (i >= 0) && (rr[i] == 0); i--)
1664                 ;
1665         ret->len = i + 1;
1624         ret->len = i+1;

1667         return (BIG_OK);
1668 }
______unchanged_portion_omitted_


1789 #ifdef  USE_FLOATING_POINT
1790 #define big_modexp_ncp_float    big_modexp_ncp_sw
1791 #else
1792 #define big_modexp_ncp_int      big_modexp_ncp_sw
1793 #endif

1795 #define MAX_EXP_BIT_GROUP_SIZE 6
1796 #define APOWERS_MAX_SIZE (1 << (MAX_EXP_BIT_GROUP_SIZE - 1))

1798 /* ARGSUSED */
1799 static BIG_ERR_CODE
1800 big_modexp_ncp_int(BIGNUM *result, BIGNUM *ma, BIGNUM *e, BIGNUM *n,
1801     BIGNUM *tmp, BIG_CHUNK_TYPE n0)

1803 {
1804         BIGNUM          apowers[APOWERS_MAX_SIZE];
1805         BIGNUM          tmp1;
1806         BIG_CHUNK_TYPE  tmp1value[BIGTMPSIZE];
```

```
1807         int             i, j, k, l, m, p;
1808         uint32_t        bit, bitind, bitcount, groupbits, apowerssize;
1809         uint32_t        nbits;
1767         int             bit, bitind, bitcount, groupbits, apowerssize;
1768         int             nbits;
1810         BIG_ERR_CODE    err;

1812         nbits = big_numbits(e);
1813         if (nbits < 50) {
1814                 groupbits = 1;
1815                 apowerssize = 1;
1816         } else {
1817                 groupbits = MAX_EXP_BIT_GROUP_SIZE;
1818                 apowerssize = 1 << (groupbits - 1);
1819         }


1822         if ((err = big_init1(&tmp1, 2 * n->len + 1,
1823             tmp1value, arraysize(tmp1value))) != BIG_OK) {
1824                 return (err);
1825         }

1827         /* clear the malloced bit to help cleanup */
1786         /* set the malloced bit to help cleanup */
1828         for (i = 0; i < apowerssize; i++) {
1829                 apowers[i].malloced = 0;
1830         }

1832         for (i = 0; i < apowerssize; i++) {
1833                 if ((err = big_init1(&(apowers[i]), n->len, NULL, 0)) !=
1834                     BIG_OK) {
1835                         goto ret;
1836                 }
1837         }

1839         (void) big_copy(&(apowers[0]), ma);

1841         if ((err = big_mont_mul(&tmp1, ma, ma, n, n0)) != BIG_OK) {
1842                 goto ret;
1843         }
1844         (void) big_copy(ma, &tmp1);

1846         for (i = 1; i < apowerssize; i++) {
1847                 if ((err = big_mont_mul(&tmp1, ma,
1848                     &(apowers[i - 1]), n, n0)) != BIG_OK) {
1806                     &(apowers[i-1]), n, n0)) != BIG_OK) {
1849                         goto ret;
1850                 }
1851                 (void) big_copy(&apowers[i], &tmp1);
1852         }

1854         bitind = nbits % BIG_CHUNK_SIZE;
1855         k = 0;
1856         l = 0;
1857         p = 0;
1858         bitcount = 0;
1859         for (i = nbits / BIG_CHUNK_SIZE; i >= 0; i--) {
1860                 for (j = bitind - 1; j >= 0; j--) {
1861                         bit = (e->value[i] >> j) & 1;
1862                         if ((bitcount == 0) && (bit == 0)) {
1863                                 if ((err = big_mont_mul(tmp,
1864                                     tmp, tmp, n, n0)) != BIG_OK) {
1865                                         goto ret;
1866                                 }
1867                         } else {
1868                                 bitcount++;
```



```
1869                                 p = p * 2 + bit;
1870                                 if (bit == 1) {
1871                                         k = k + l + 1;
1872                                         l = 0;
1873                                 } else {
1874                                         l++;
1875                                 }
1876                                 if (bitcount == groupbits) {
1877                                         for (m = 0; m < k; m++) {
1878                                                 if ((err = big_mont_mul(tmp,
1879                                                     tmp, tmp, n, n0)) !=
1880                                                     BIG_OK) {
1881                                                         goto ret;
1882                                                 }
1883                                         }
1884                                         if ((err = big_mont_mul(tmp, tmp,
1885                                             &(apowers[p >> (l + 1)]),
1886                                             n, n0)) != BIG_OK) {
1887                                                 goto ret;
1888                                         }
1889                                         for (m = 0; m < l; m++) {
1890                                                 if ((err = big_mont_mul(tmp,
1891                                                     tmp, tmp, n, n0)) !=
1892                                                     BIG_OK) {
1893                                                         goto ret;
1894                                                 }
1895                                         }
1896                                         k = 0;
1897                                         l = 0;
1898                                         p = 0;
1899                                         bitcount = 0;
1900                                 }
1901                         }
1902                 }
1903                 bitind = BIG_CHUNK_SIZE;
1904         }

1906         for (m = 0; m < k; m++) {
1907                 if ((err = big_mont_mul(tmp, tmp, tmp, n, n0)) != BIG_OK) {
1908                         goto ret;
1909                 }
1910         }
1911         if (p != 0) {
1912                 if ((err = big_mont_mul(tmp, tmp,
1913                     &(apowers[p >> (l + 1)]), n, n0)) != BIG_OK) {
1914                         goto ret;
1915                 }
1916         }
1917         for (m = 0; m < l; m++) {
1918                 if ((err = big_mont_mul(result, tmp, tmp, n, n0)) != BIG_OK) {
1919                         goto ret;
1920                 }
1921         }

1923 ret:
1924         for (i = apowerssize - 1; i >= 0; i--) {
1925                 big_finish(&(apowers[i]));
1926         }
1927         big_finish(&tmp1);

1929         return (err);
1930 }


1933 #ifdef USE_FLOATING_POINT
```

```
1935 #ifdef _KERNEL

1937 #include <sys/sysmacros.h>
1938 #include <sys/regset.h>
1939 #include <sys/fpu/fpusystm.h>

1941 /* the alignment for block stores to save fp registers */
1942 #define FPR_ALIGN       (64)

1944 extern void big_savefp(kfpu_t *);
1945 extern void big_restorefp(kfpu_t *);

1947 #endif /* _KERNEL */

1949 /*
1950  * This version makes use of floating point for performance
1951  */
1952 static BIG_ERR_CODE
1953 big_modexp_ncp_float(BIGNUM *result, BIGNUM *ma, BIGNUM *e, BIGNUM *n,
1954     BIGNUM *tmp, BIG_CHUNK_TYPE n0)
1955 {

1957         int             i, j, k, l, m, p;
1958         uint32_t        bit, bitind, bitcount, nlen;
1915         int             i, j, k, l, m, p, bit, bitind, bitcount, nlen;
1959         double          dn0;
1960         double          *dn, *dt, *d16r, *d32r;
1961         uint32_t        *nint, *prod;
1962         double          *apowers[APOWERS_MAX_SIZE];
1963         uint32_t        nbits, groupbits, apowerssize;
1920         int             nbits, groupbits, apowerssize;
1964         BIG_ERR_CODE    err = BIG_OK;

1966 #ifdef _KERNEL
1967         uint8_t fpua[sizeof (kfpu_t) + FPR_ALIGN];
1968         kfpu_t *fpu;

1970 #ifdef DEBUG
1971         if (!fpu_exists)
1972                 return (BIG_GENERAL_ERR);
1973 #endif

1975         fpu =  (kfpu_t *)P2ROUNDUP((uintptr_t)fpua, FPR_ALIGN);
1976         big_savefp(fpu);

1978 #endif /* _KERNEL */

1980         nbits = big_numbits(e);
1981         if (nbits < 50) {
1982                 groupbits = 1;
1983                 apowerssize = 1;
1984         } else {
1985                 groupbits = MAX_EXP_BIT_GROUP_SIZE;
1986                 apowerssize = 1 << (groupbits - 1);
1987         }

1989         nlen = (BIG_CHUNK_SIZE / 32) * n->len;
1990         dn0 = (double)(n0 & 0xffff);

1992         dn = dt = d16r = d32r = NULL;
1993         nint = prod = NULL;
1994         for (i = 0; i < apowerssize; i++) {
1995                 apowers[i] = NULL;
1996         }

1998         if ((dn = big_malloc(nlen * sizeof (double))) == NULL) {
```



```
1999                 err = BIG_NO_MEM;
2000                 goto ret;
2001         }
2002         if ((dt = big_malloc((4 * nlen + 2) * sizeof (double))) == NULL) {
2003                 err = BIG_NO_MEM;
2004                 goto ret;
2005         }
2006         if ((nint = big_malloc(nlen * sizeof (uint32_t))) == NULL) {
2007                 err = BIG_NO_MEM;
2008                 goto ret;
2009         }
2010         if ((prod = big_malloc((nlen + 1) * sizeof (uint32_t))) == NULL) {
2011                 err = BIG_NO_MEM;
2012                 goto ret;
2013         }
2014         if ((d16r = big_malloc((2 * nlen + 1) * sizeof (double))) == NULL) {
2015                 err = BIG_NO_MEM;
2016                 goto ret;
2017         }
2018         if ((d32r = big_malloc(nlen * sizeof (double))) == NULL) {
2019                 err = BIG_NO_MEM;
2020                 goto ret;
2021         }
2022         for (i = 0; i < apowerssize; i++) {
2023                 if ((apowers[i] = big_malloc((2 * nlen + 1) *
2024                     sizeof (double))) == NULL) {
2025                         err = BIG_NO_MEM;
2026                         goto ret;
2027                 }
2028         }

2030 #if (BIG_CHUNK_SIZE == 32)
2031         for (i = 0; i < ma->len; i++) {
2032                 nint[i] = ma->value[i];
2033         }
2034         for (; i < nlen; i++) {
2035                 nint[i] = 0;
2036         }
2037 #else
2038         for (i = 0; i < ma->len; i++) {
2039                 nint[2 * i] = (uint32_t)(ma->value[i] & 0xffffffffULL);
2040                 nint[2 * i + 1] = (uint32_t)(ma->value[i] >> 32);
2041         }
2042         for (i = ma->len * 2; i < nlen; i++) {
2043                 nint[i] = 0;
2044         }
2045 #endif
2046         conv_i32_to_d32_and_d16(d32r, apowers[0], nint, nlen);

2048 #if (BIG_CHUNK_SIZE == 32)
2049         for (i = 0; i < n->len; i++) {
2050                 nint[i] = n->value[i];
2051         }
2052         for (; i < nlen; i++) {
2053                 nint[i] = 0;
2054         }
2055 #else
2056         for (i = 0; i < n->len; i++) {
2057                 nint[2 * i] = (uint32_t)(n->value[i] & 0xffffffffULL);
2058                 nint[2 * i + 1] = (uint32_t)(n->value[i] >> 32);
2059         }
2060         for (i = n->len * 2; i < nlen; i++) {
2061                 nint[i] = 0;
2062         }
2063 #endif
2064         conv_i32_to_d32(dn, nint, nlen);
```

```
2066         mont_mulf_noconv(prod, d32r, apowers[0], dt, dn, nint, nlen, dn0);
2067         conv_i32_to_d32(d32r, prod, nlen);
2068         for (i = 1; i < apowerssize; i++) {
2069                 mont_mulf_noconv(prod, d32r, apowers[i - 1],
2070                     dt, dn, nint, nlen, dn0);
2071                 conv_i32_to_d16(apowers[i], prod, nlen);
2072         }

2074 #if (BIG_CHUNK_SIZE == 32)
2075         for (i = 0; i < tmp->len; i++) {
2076                 prod[i] = tmp->value[i];
2077         }
2078         for (; i < nlen + 1; i++) {
2079                 prod[i] = 0;
2080         }
2081 #else
2082         for (i = 0; i < tmp->len; i++) {
2083                 prod[2 * i] = (uint32_t)(tmp->value[i] & 0xffffffffULL);
2084                 prod[2 * i + 1] = (uint32_t)(tmp->value[i] >> 32);
2085         }
2086         for (i = tmp->len * 2; i < nlen + 1; i++) {
2087                 prod[i] = 0;
2088         }
2089 #endif

2091         bitind = nbits % BIG_CHUNK_SIZE;
2092         k = 0;
2093         l = 0;
2094         p = 0;
2095         bitcount = 0;
2096         for (i = nbits / BIG_CHUNK_SIZE; i >= 0; i--) {
2097                 for (j = bitind - 1; j >= 0; j--) {
2098                         bit = (e->value[i] >> j) & 1;
2099                         if ((bitcount == 0) && (bit == 0)) {
2100                                 conv_i32_to_d32_and_d16(d32r, d16r,
2101                                     prod, nlen);
2102                                 mont_mulf_noconv(prod, d32r, d16r,
2103                                     dt, dn, nint, nlen, dn0);
2104                         } else {
2105                                 bitcount++;
2106                                 p = p * 2 + bit;
2107                                 if (bit == 1) {
2108                                         k = k + l + 1;
2109                                         l = 0;
2110                                 } else {
2111                                         l++;
2112                                 }
2113                                 if (bitcount == groupbits) {
2114                                         for (m = 0; m < k; m++) {
2115                                                 conv_i32_to_d32_and_d16(d32r,
2116                                                     d16r, prod, nlen);
2117                                                 mont_mulf_noconv(prod, d32r,
2118                                                     d16r, dt, dn, nint,
2119                                                     nlen, dn0);
2120                                         }
2121                                         conv_i32_to_d32(d32r, prod, nlen);
2122                                         mont_mulf_noconv(prod, d32r,
2123                                             apowers[p >> (l + 1)],
2080                                             apowers[p >> (l+1)],
2124                                             dt, dn, nint, nlen, dn0);
2125                                         for (m = 0; m < l; m++) {
2126                                                 conv_i32_to_d32_and_d16(d32r,
2127                                                     d16r, prod, nlen);
2128                                                 mont_mulf_noconv(prod, d32r,
2129                                                     d16r, dt, dn, nint,
```



```
2130                                                     nlen, dn0);
2131                                         }
2132                                         k = 0;
2133                                         l = 0;
2134                                         p = 0;
2135                                         bitcount = 0;
2136                                 }
2137                         }
2138                 }
2139                 bitind = BIG_CHUNK_SIZE;
2140         }

2142         for (m = 0; m < k; m++) {
2143                 conv_i32_to_d32_and_d16(d32r, d16r, prod, nlen);
2144                 mont_mulf_noconv(prod, d32r, d16r, dt, dn, nint, nlen, dn0);
2145         }
2146         if (p != 0) {
2147                 conv_i32_to_d32(d32r, prod, nlen);
2148                 mont_mulf_noconv(prod, d32r, apowers[p >> (l + 1)],
2149                     dt, dn, nint, nlen, dn0);
2150         }
2151         for (m = 0; m < l; m++) {
2152                 conv_i32_to_d32_and_d16(d32r, d16r, prod, nlen);
2153                 mont_mulf_noconv(prod, d32r, d16r, dt, dn, nint, nlen, dn0);
2154         }

2156 #if (BIG_CHUNK_SIZE == 32)
2157         for (i = 0; i < nlen; i++) {
2158                 result->value[i] = prod[i];
2159         }
2160         for (i = nlen - 1; (i > 0) && (prod[i] == 0); i--)
2161                 ;
2162 #else
2163         for (i = 0; i < nlen / 2; i++) {
2164                 result->value[i] = (uint64_t)(prod[2 * i]) +
2165                     (((uint64_t)(prod[2 * i + 1])) << 32);
2166         }
2167         for (i = nlen / 2 - 1; (i > 0) && (result->value[i] == 0); i--)
2168                 ;
2169 #endif
2170         result->len = i + 1;

2172 ret:
2173         for (i = apowerssize - 1; i >= 0; i--) {
2174                 if (apowers[i] != NULL)
2175                         big_free(apowers[i], (2 * nlen + 1) * sizeof (double));
2176         }
2177         if (d32r != NULL) {
2178                 big_free(d32r, nlen * sizeof (double));
2179         }
2180         if (d16r != NULL) {
2181                 big_free(d16r, (2 * nlen + 1) * sizeof (double));
2182         }
2183         if (prod != NULL) {
2184                 big_free(prod, (nlen + 1) * sizeof (uint32_t));
2185         }
2186         if (nint != NULL) {
2187                 big_free(nint, nlen * sizeof (uint32_t));
2188         }
2189         if (dt != NULL) {
2190                 big_free(dt, (4 * nlen + 2) * sizeof (double));
2191         }
2192         if (dn != NULL) {
2193                 big_free(dn, nlen * sizeof (double));
2194         }
```

```
#ifdef _KERNEL
        big_restorefp(fpu);
#endif

        return (err);
}

#endif /* USE_FLOATING_POINT */


BIG_ERR_CODE
big_modexp_ext(BIGNUM *result, BIGNUM *a, BIGNUM *e, BIGNUM *n, BIGNUM *n_rr,
    big_modexp_ncp_info_t *info)
{
        BIGNUM          ma, tmp, rr;
        BIG_CHUNK_TYPE  mavalue[BIGTMPSIZE];
        BIG_CHUNK_TYPE  tmpvalue[BIGTMPSIZE];
        BIG_CHUNK_TYPE  rrvalue[BIGTMPSIZE];
        BIG_ERR_CODE    err;
        BIG_CHUNK_TYPE  n0;

        if ((err = big_init1(&ma, n->len, mavalue, arraysize(mavalue))) !=
            BIG_OK) {
                return (err);
        }
        ma.len = 1;
        ma.value[0] = 0;

        if ((err = big_init1(&tmp, 2 * n->len + 1,
            tmpvalue, arraysize(tmpvalue))) != BIG_OK) {
                goto ret1;
        }

        /* clear the malloced bit to help cleanup */
        /* set the malloced bit to help cleanup */
        rr.malloced = 0;

        if (n_rr == NULL) {
                if ((err = big_init1(&rr, 2 * n->len + 1,
                    rrvalue, arraysize(rrvalue))) != BIG_OK) {
                        goto ret2;
                }
                if (big_mont_rr(&rr, n) != BIG_OK) {
                        goto ret;
                }
                n_rr = &rr;
        }

        n0 = big_n0(n->value[0]);

        if (big_cmp_abs(a, n) > 0) {
                if ((err = big_div_pos(NULL, &ma, a, n)) != BIG_OK) {
                        goto ret;
                }
                err = big_mont_conv(&ma, &ma, n, n0, n_rr);
        } else {
                err = big_mont_conv(&ma, a, n, n0, n_rr);
        }
        if (err != BIG_OK) {
                goto ret;
        }

        tmp.len = 1;
        tmp.value[0] = 1;
        if ((err = big_mont_conv(&tmp, &tmp, n, n0, n_rr)) != BIG_OK) {
                goto ret;
```



```
        }

        if ((info != NULL) && (info->func != NULL)) {
                err = (*(info->func))(&tmp, &ma, e, n, &tmp, n0,
                    info->ncp, info->reqp);
        } else {
                err = big_modexp_ncp_sw(&tmp, &ma, e, n, &tmp, n0);
        }
        if (err != BIG_OK) {
                goto ret;
        }

        ma.value[0] = 1;
        ma.len = 1;
        if ((err = big_mont_mul(&tmp, &tmp, &ma, n, n0)) != BIG_OK) {
                goto ret;
        }
        err = big_copy(result, &tmp);

ret:
        if (rr.malloced) {
                big_finish(&rr);
        }
ret2:
        big_finish(&tmp);
ret1:
        big_finish(&ma);

        return (err);
}
```

*unchanged_portion_omitted_*

```
static BIG_CHUNK_TYPE onearr[1] = {(BIG_CHUNK_TYPE)1};
BIGNUM big_One = {1, 1, 1, 0, onearr};

static BIG_CHUNK_TYPE twoarr[1] = {(BIG_CHUNK_TYPE)2};
BIGNUM big_Two = {1, 1, 1, 0, twoarr};

static BIG_CHUNK_TYPE fourarr[1] = {(BIG_CHUNK_TYPE)4};
static BIGNUM big_Four = {1, 1, 1, 0, fourarr};


BIG_ERR_CODE
big_sqrt_pos(BIGNUM *result, BIGNUM *n)
{
        BIGNUM          *high, *low, *mid, *t;
        BIGNUM          t1, t2, t3, prod;
        BIG_CHUNK_TYPE  t1value[BIGTMPSIZE];
        BIG_CHUNK_TYPE  t2value[BIGTMPSIZE];
        BIG_CHUNK_TYPE  t3value[BIGTMPSIZE];
        BIG_CHUNK_TYPE  prodvalue[BIGTMPSIZE];
        int             i, diff;
        uint32_t        nbits, nrootbits, highbits;
        int             i, nbits, diff, nrootbits, highbits;
        BIG_ERR_CODE    err;

        nbits = big_numbits(n);

        if ((err = big_init1(&t1, n->len + 1,
            t1value, arraysize(t1value))) != BIG_OK)
                return (err);
        if ((err = big_init1(&t2, n->len + 1,
            t2value, arraysize(t2value))) != BIG_OK)
                goto ret1;
        if ((err = big_init1(&t3, n->len + 1,
```

```
2441             t3value, arraysize(t3value))) != BIG_OK)
2442                 goto ret2;
2443         if ((err = big_init1(&prod, n->len + 1,
2444             prodvalue, arraysize(prodvalue))) != BIG_OK)
2445                 goto ret3;

2447         nrootbits = (nbits + 1) / 2;
2448         t1.len = t2.len = t3.len = (nrootbits - 1) / BIG_CHUNK_SIZE + 1;
2449         for (i = 0; i < t1.len; i++) {
2450                 t1.value[i] = 0;
2451                 t2.value[i] = BIG_CHUNK_ALLBITS;
2452         }
2453         highbits = nrootbits - BIG_CHUNK_SIZE * (t1.len - 1);
2454         if (highbits == BIG_CHUNK_SIZE) {
2455                 t1.value[t1.len - 1] = BIG_CHUNK_HIGHBIT;
2456                 t2.value[t2.len - 1] = BIG_CHUNK_ALLBITS;
2457         } else {
2458                 t1.value[t1.len - 1] = (BIG_CHUNK_TYPE)1 << (highbits - 1);
2459                 t2.value[t2.len - 1] = 2 * t1.value[t1.len - 1] - 1;
2460         }

2462         high = &t2;
2463         low = &t1;
2464         mid = &t3;

2466         if ((err = big_mul(&prod, high, high)) != BIG_OK) {
2467                 goto ret;
2468         }
2469         diff = big_cmp_abs(&prod, n);
2470         if (diff <= 0) {
2471                 err = big_copy(result, high);
2472                 goto ret;
2473         }

2475         (void) big_sub_pos(mid, high, low);
2476         while (big_cmp_abs(&big_One, mid) != 0) {
2477                 (void) big_add_abs(mid, high, low);
2478                 (void) big_half_pos(mid, mid);
2479                 if ((err = big_mul(&prod, mid, mid)) != BIG_OK)
2480                         goto ret;
2481                 diff = big_cmp_abs(&prod, n);
2482                 if (diff > 0) {
2483                         t = high;
2484                         high = mid;
2485                         mid = t;
2486                 } else if (diff < 0) {
2487                         t = low;
2488                         low = mid;
2489                         mid = t;
2490                 } else {
2491                         err = big_copy(result, low);
2492                         goto ret;
2493                 }
2494                 (void) big_sub_pos(mid, high, low);
2495         }

2497         err = big_copy(result, low);
2498 ret:
2499         if (prod.malloced) big_finish(&prod);
2500 ret3:
2501         if (t3.malloced) big_finish(&t3);
2502 ret2:
2503         if (t2.malloced) big_finish(&t2);
2504 ret1:
2505         if (t1.malloced) big_finish(&t1);
```



```
2507         return (err);
2508 }
_____unchanged_portion_omitted_


2596 BIG_ERR_CODE
2597 big_Lucas(BIGNUM *Lkminus1, BIGNUM *Lk, BIGNUM *p, BIGNUM *k, BIGNUM *n)
2598 {
2599         int             i;
2600         uint32_t        m, w;
2554         int             m, w, i;
2601         BIG_CHUNK_TYPE  bit;
2602         BIGNUM          ki, tmp, tmp2;
2603         BIG_CHUNK_TYPE  kivalue[BIGTMPSIZE];
2604         BIG_CHUNK_TYPE  tmpvalue[BIGTMPSIZE];
2605         BIG_CHUNK_TYPE  tmp2value[BIGTMPSIZE];
2606         BIG_ERR_CODE    err;

2608         if (big_cmp_abs(k, &big_One) == 0) {
2609                 (void) big_copy(Lk, p);
2610                 (void) big_copy(Lkminus1, &big_Two);
2611                 return (BIG_OK);
2612         }

2614         if ((err = big_init1(&ki, k->len + 1,
2615             kivalue, arraysize(kivalue))) != BIG_OK)
2616                 return (err);

2618         if ((err = big_init1(&tmp, 2 * n->len + 1,
2572         if ((err = big_init1(&tmp, 2 * n->len +1,
2619             tmpvalue, arraysize(tmpvalue))) != BIG_OK)
2620                 goto ret1;

2622         if ((err = big_init1(&tmp2, n->len,
2623             tmp2value, arraysize(tmp2value))) != BIG_OK)
2624                 goto ret2;

2626         m = big_numbits(k);
2627         ki.len = (m - 1) / BIG_CHUNK_SIZE + 1;
2628         w = (m - 1) / BIG_CHUNK_SIZE;
2629         bit = (BIG_CHUNK_TYPE)1 << ((m - 1) % BIG_CHUNK_SIZE);
2630         for (i = 0; i < ki.len; i++) {
2631                 ki.value[i] = 0;
2632         }
2633         ki.value[ki.len - 1] = bit;
2634         if (big_cmp_abs(k, &ki) != 0) {
2635                 (void) big_double(&ki, &ki);
2636         }
2637         (void) big_sub_pos(&ki, &ki, k);

2639         (void) big_copy(Lk, p);
2640         (void) big_copy(Lkminus1, &big_Two);

2642         for (i = 0; i < m; i++) {
2643                 if ((err = big_mul(&tmp, Lk, Lkminus1)) != BIG_OK) {
2644                         goto ret;
2645                 }
2646                 (void) big_add_abs(&tmp, &tmp, n);
2647                 (void) big_sub_pos(&tmp, &tmp, p);
2648                 if ((err = big_div_pos(NULL, &tmp2, &tmp, n)) != BIG_OK) {
2649                         goto ret;
2650                 }
2651                 if ((ki.value[w] & bit) != 0) {
2652                         if ((err = big_mul(&tmp, Lkminus1, Lkminus1)) !=
2653                             BIG_OK) {
2654                                 goto ret;
```

```
2655                        }
2656                        (void) big_add_abs(&tmp, &tmp, n);
2657                        (void) big_sub_pos(&tmp, &tmp, &big_Two);
2658                        if ((err = big_div_pos(NULL, Lkminus1, &tmp, n)) !=
2659                            BIG_OK) {
2660                                goto ret;
2661                        }
2662                        (void) big_copy(Lk, &tmp2);
2663                } else {
2664                        if ((err = big_mul(&tmp, Lk, Lk)) != BIG_OK) {
2665                                goto ret;
2666                        }
2667                        (void) big_add_abs(&tmp, &tmp, n);
2668                        (void) big_sub_pos(&tmp, &tmp, &big_Two);
2669                        if ((err = big_div_pos(NULL, Lk, &tmp, n)) != BIG_OK) {
2670                                goto ret;
2671                        }
2672                        (void) big_copy(Lkminus1, &tmp2);
2673                }
2674                bit = bit >> 1;
2675                if (bit == 0) {
2676                        bit = BIG_CHUNK_HIGHBIT;
2677                        w--;
2678                }
2679        }

2681        err = BIG_OK;

2683 ret:
2684        if (tmp2.malloced) big_finish(&tmp2);
2685 ret2:
2686        if (tmp.malloced) big_finish(&tmp);
2687 ret1:
2688        if (ki.malloced) big_finish(&ki);

2690        return (err);
2691 }
```

*unchanged_portion_omitted_*

```
2828 #define SIEVESIZE 1000

2784 uint32_t smallprimes[] =
2785 {
2786 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
2787 51, 53, 59, 61, 67, 71, 73, 79, 83, 89, 91, 97
2788 };


2831 BIG_ERR_CODE
2832 big_nextprime_pos_ext(BIGNUM *result, BIGNUM *n, big_modexp_ncp_info_t *info)
2833 {
2834        static const uint32_t smallprimes[] = {
2835            3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
2836            51, 53, 59, 61, 67, 71, 73, 79, 83, 89, 91, 97 };
2837        BIG_ERR_CODE    err;
2838        int             sieve[SIEVESIZE];
2839        int             i;
2840        uint32_t        off, p;

2842        if ((err = big_copy(result, n)) != BIG_OK) {
2843                return (err);
2844        }
2845        result->value[0] |= 1;
2846        /* CONSTCOND */
2847        while (1) {
```



```
2848                for (i = 0; i < SIEVESIZE; i++) sieve[i] = 0;
2849                for (i = 0;
2850                    i < sizeof (smallprimes) / sizeof (smallprimes[0]); i++) {
2851                        p = smallprimes[i];
2852                        off = big_modhalf_pos(result, p);
2853                        off = p - off;
2854                        if ((off % 2) == 1) {
2855                                off = off + p;
2856                        }
2857                        off = off / 2;
2858                        while (off < SIEVESIZE) {
2859                                sieve[off] = 1;
2860                                off = off + p;
2861                        }
2862                }

2864                for (i = 0; i < SIEVESIZE; i++) {
2865                        if (sieve[i] == 0) {
2866                                err = big_isprime_pos_ext(result, info);
2867                                if (err != BIG_FALSE) {
2868                                        if (err != BIG_TRUE) {
2869                                                return (err);
2870                                        } else {
2871                                                goto out;
2872                                        }
2873                                }

2875                        }
2876                        if ((err = big_add_abs(result, result, &big_Two)) !=
2877                            BIG_OK) {
2878                                return (err);
2879                        }
2880                }
2881        }

2883 out:
2884        return (BIG_OK);
2885 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    8132 Wed Feb 25 22:20:00 2009
new/usr/src/common/bignum/mont_mulf.c
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*
 * If compiled without -DRF_INLINE_MACROS then needs -lm at link time
 * If compiled with -DRF_INLINE_MACROS then needs conv.il at compile time
 * (i.e. cc <compiler_flags> -DRF_INLINE_MACROS conv.il mont_mulf.c )
 * (i.e. cc <compileer_flags> -DRF_INLINE_MACROS conv.il mont_mulf.c )
 */

#include <sys/types.h>
#include <math.h>

static const double TwoTo16 = 65536.0;
static const double TwoToMinus16 = 1.0/65536.0;
static const double Zero = 0.0;
static const double TwoTo32 = 65536.0 * 65536.0;
static const double TwoToMinus32 = 1.0 / (65536.0 * 65536.0);

#ifdef RF_INLINE_MACROS

double upper32(double);
double lower32(double, double);
double mod(double, double, double);

#else

static double
upper32(double x)
{
        return (floor(x * TwoToMinus32));
```



```
}
______unchanged_portion_omitted_


void
conv_d16_to_i32(uint32_t *i32, double *d16, int64_t *tmp, int ilen)
{
        int i;
        int64_t t, t1,          /* Using int64_t and not uint64_t */
        int64_t t, t1,          /* using int64_t and not uint64_t */
            a, b, c, d;         /* because more efficient code is */
                                /* generated this way, and there  */
                                /* is no overflow.  */
                                /* is no overflow  */
        t1 = 0;
        a = (int64_t)d16[0];
        b = (int64_t)d16[1];
        for (i = 0; i < ilen - 1; i++) {
                c = (int64_t)d16[2 * i + 2];
                t1 += a & 0xffffffff;
                t = (a >> 32);
                d = (int64_t)d16[2 * i + 3];
                t1 += (b & 0xffff) << 16;
                t += (b >> 16) + (t1 >> 32);
                i32[i] = t1 & 0xffffffff;
                t1 = t;
                a = c;
                b = d;
        }
        t1 += a & 0xffffffff;
        t = (a >> 32);
        t1 += (b & 0xffff) << 16;
        i32[i] = t1 & 0xffffffff;
}
______unchanged_portion_omitted_

#endif


void
conv_i32_to_d32_and_d16(double *d32, double *d16, uint32_t *i32, int len)
{
        int i;
        uint32_t a;

#pragma pipeloop(0)
        for (i = 0; i < len - 3; i += 4) {
                i16_to_d16_and_d32x4(&TwoToMinus16, &TwoTo16, &Zero,
                    &(d16[2*i]), &(d32[i]), (float *)(&(i32[i])));
                                        &(d16[2*i]), &(d32[i]),
                                        (float *)(&(i32[i])));
        }
        for (; i < len; i++) {
                a = i32[i];
                d32[i] = (double)(i32[i]);
                d16[2 * i] = (double)(a & 0xffff);
                d16[2 * i + 1] = (double)(a >> 16);
        }
}
______unchanged_portion_omitted_


/*
 * the lengths of the input arrays should be at least the following:
 * result[nlen+1], dm1[nlen], dm2[2*nlen+1], dt[4*nlen+2], dn[nlen], nint[nlen]
 * all of them should be different from one another
```

```
 */
void mont_mulf_noconv(uint32_t *result,
                      double *dm1, double *dm2, double *dt,
                      double *dn, uint32_t *nint,
                      int nlen, double dn0)
{
        int i, j, jj;
        double digit, m2j, a, b;
        double *pdm1, *pdm2, *pdn, *pdtj, pdn_0, pdm1_0;

        pdm1 = &(dm1[0]);
        pdm2 = &(dm2[0]);
        pdn = &(dn[0]);
        pdm2[2 * nlen] = Zero;

        if (nlen != 16) {
                for (i = 0; i < 4 * nlen + 2; i++)
                        dt[i] = Zero;
                a = dt[0] = pdm1[0] * pdm2[0];
                digit = mod(lower32(a, Zero) * dn0, TwoToMinus16, TwoTo16);

                pdtj = &(dt[0]);
                for (j = jj = 0; j < 2 * nlen; j++, jj++, pdtj++) {
                        m2j = pdm2[j];
                        a = pdtj[0] + pdn[0] * digit;
                        b = pdtj[1] + pdm1[0] * pdm2[j + 1] + a * TwoToMinus16;
                        pdtj[1] = b;

#pragma pipeloop(0)
                        for (i = 1; i < nlen; i++) {
                                pdtj[2 * i] += pdm1[i] * m2j + pdn[i] * digit;
                        }
                        if (jj == 30) {
                                cleanup(dt, j / 2 + 1, 2 * nlen + 1);
                                jj = 0;
                        }

                        digit = mod(lower32(b, Zero) * dn0,
                            TwoToMinus16, TwoTo16);
                }
        } else {
                a = dt[0] = pdm1[0] * pdm2[0];

                dt[65] = dt[64] = dt[63] = dt[62] = dt[61] = dt[60] =
                    dt[59] = dt[58] = dt[57] = dt[56] = dt[55] =
                    dt[54] = dt[53] = dt[52] = dt[51] = dt[50] =
                    dt[49] = dt[48] = dt[47] = dt[46] = dt[45] =
                    dt[44] = dt[43] = dt[42] = dt[41] = dt[40] =
                    dt[39] = dt[38] = dt[37] = dt[36] = dt[35] =
                    dt[34] = dt[33] = dt[32] = dt[31] = dt[30] =
                    dt[29] = dt[28] = dt[27] = dt[26] = dt[25] =
                    dt[24] = dt[23] = dt[22] = dt[21] = dt[20] =
                    dt[19] = dt[18] = dt[17] = dt[16] = dt[15] =
                    dt[14] = dt[13] = dt[12] = dt[11] = dt[10] =
                    dt[9] = dt[8] = dt[7] = dt[6] = dt[5] = dt[4] =
                    dt[3] = dt[2] = dt[1] = Zero;

                pdn_0 = pdn[0];
                pdm1_0 = pdm1[0];

                digit = mod(lower32(a, Zero) * dn0, TwoToMinus16, TwoTo16);
                pdtj = &(dt[0]);

                for (j = 0; j < 32; j++, pdtj++) {

                        m2j = pdm2[j];
```



```
                        a = pdtj[0] + pdn_0 * digit;
                        b = pdtj[1] + pdm1_0 * pdm2[j + 1] + a * TwoToMinus16;
                        pdtj[1] = b;

                        pdtj[2] += pdm1[1] *m2j + pdn[1] * digit;
                        pdtj[4] += pdm1[2] *m2j + pdn[2] * digit;
                        pdtj[6] += pdm1[3] *m2j + pdn[3] * digit;
                        pdtj[8] += pdm1[4] *m2j + pdn[4] * digit;
                        pdtj[10] += pdm1[5] *m2j + pdn[5] * digit;
                        pdtj[12] += pdm1[6] *m2j + pdn[6] * digit;
                        pdtj[14] += pdm1[7] *m2j + pdn[7] * digit;
                        pdtj[16] += pdm1[8] *m2j + pdn[8] * digit;
                        pdtj[18] += pdm1[9] *m2j + pdn[9] * digit;
                        pdtj[20] += pdm1[10] *m2j + pdn[10] * digit;
                        pdtj[22] += pdm1[11] *m2j + pdn[11] * digit;
                        pdtj[24] += pdm1[12] *m2j + pdn[12] * digit;
                        pdtj[26] += pdm1[13] *m2j + pdn[13] * digit;
                        pdtj[28] += pdm1[14] *m2j + pdn[14] * digit;
                        pdtj[30] += pdm1[15] *m2j + pdn[15] * digit;
                        /* no need for cleanup, cannot overflow */
                        /* no need for cleenup, cannot overflow */
                        digit = mod(lower32(b, Zero) * dn0,
                            TwoToMinus16, TwoTo16);
                }
        }

        conv_d16_to_i32(result, dt + 2 * nlen, (int64_t *)dt, nlen + 1);
        adjust_montf_result(result, nint, nlen);
}
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2143 Wed Feb 25 22:20:12 2009
new/usr/src/lib/pkcs11/libsoftcrypto/amd64/Makefile
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
# lib/pkcs11/libsoftcrypto/amd64/Makefile
#

LIBRARY=        libsoftcrypto.a
VERS= .1

AES_PSM_OBJS= aes_amd64.o aeskey.o
AES_PSM_SRC= $(AES_DIR)/$(MACH64)/aes_amd64.s $(AES_DIR)/$(MACH64)/aeskey.c

ARCFOUR_PSM_OBJS= arcfour-x86_64.o
ARCFOUR_PSM_SRC= arcfour-x86_64.s

BIGNUM_PSM_OBJS= bignum_amd64.o bignum_amd64_asm.o
BIGNUM_PSM_SRC= $(BIGNUM_DIR)/$(MACH64)/bignum_amd64.c \
                $(BIGNUM_DIR)/$(MACH64)/bignum_amd64_asm.s


include ../Makefile.com
include $(SRC)/lib/Makefile.lib.64

CFLAGS          += -xO4 -xcrossfile
BIGNUM_FLAGS    += -DPSR_MUL
LINTFLAGS64     += $(BIGNUM_FLAGS) $(AES_FLAGS)

CLEANFILES      += arcfour-x86_64.s

LDLIBS  += -lc
LIBS += $(LINTLIB)

install: all $(ROOTLIBS64) $(ROOTLINKS64) $(ROOTLINT64)

arcfour-x86_64.s:       $(ARCFOUR_DIR)/amd64/arcfour-x86_64.pl
        $(PERL) $? $@
```



```
pics/%.o:       $(AES_DIR)/$(MACH64)/%.c
        $(COMPILE.c) $(AES_FLAGS) -o $@ $<
        $(POST_PROCESS_O)

pics/%.o:       $(AES_DIR)/$(MACH64)/%.s
        $(COMPILE.s) $(AES_FLAGS) -o $@ $<
        $(POST_PROCESS_O)

pics/%.o:       $(BIGNUM_DIR)/$(MACH64)/%.c
        $(COMPILE.c) $(BIGNUM_FLAGS) -o $@ $<
        $(POST_PROCESS_O)

pics/%.o:       $(BIGNUM_DIR)/$(MACH64)/%.s
        $(COMPILE64.s) $(BIGNUM_FLAGS) -o $@ $<
        $(POST_PROCESS_O)

pics/%.o:       arcfour-x86_64.s
        $(COMPILE64.s) $(ARCFOUR_FLAGS) -o $@ $<
        $(POST_PROCESS_O)
```

```
**********************************************************
    2462 Wed Feb 25 22:20:22 2009
new/usr/src/uts/intel/bignum/Makefile.64
6799218 RSA using Solaris Kernel Crypto framework lagging behind OpenSSL
5016936 bignumimpl:big_mul: potential memory leak
6810280 panic from bignum module: vmem_xalloc(): size == 0
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#ident  "%Z%%M% %I%     %E% SMI"
#
#       Configuration and targets for bignum module
#       specific to AMD 64-bit architecture, amd64.
#
# Bignum configuration (BIGNUM_CFG):
#   PSR_MUL:
#       There is a processor-specific implementation bignum multiply functions
#   HWCAP:
#       There are multiple implementations of bignum functions, and the
#       appropriate one must be chosen at run time, based on testing
#       hardware capabilites.
#
# -DPSR_MUL:
# For AMD64, there is a processor-specific implementation of
# the bignum multiply functions, which takes advantage of the
# 64x64->128 bit multiply instruction.
#
# -UHWCAP:
# There is only one implementation, because the 128 bit multiply using
# general-purpose registers is faster than any MMX or SSE2 implementation.

BIGNUM_CFG = -DPSR_MUL
CFLAGS  += -xO4 -xcrossfile

$(OBJS_DIR)/bignumimpl.o $(LINTS_DIR)/bignumimpl.ln := \
        CPPFLAGS += $(BIGNUM_CFG)
$(OBJS_DIR)/bignum_amd64.o $(LINTS_DIR)/bignum_amd64.ln := \
        CPPFLAGS += $(BIGNUM_CFG)

$(OBJS_DIR)/bignum_amd64.o: $(BIGNUMDIR)/amd64/bignum_amd64.c
        $(COMPILE.c) -o $@ $(BIGNUM_CFG) $(BIGNUMDIR)/amd64/bignum_amd64.c
        $(COMPILE.c) -o $@ $(BIGNUMDIR)/amd64/bignum_amd64.c
        $(CTFCONVERT_O)
```



```
        $(POST_PROCESS_O)

$(OBJS_DIR)/bignum_amd64_asm.o: $(BIGNUMDIR)/amd64/bignum_amd64_asm.s
        $(COMPILE.s) -P -o $@ $(BIGNUM_CFG) \
                $(BIGNUMDIR)/amd64/bignum_amd64_asm.s
        $(COMPILE.s) -P -o $@ $(BIGNUM_CFG) $(BIGNUMDIR)/amd64/bignum_amd64_asm.
        $(POST_PROCESS_O)

$(LINTS_DIR)/bignum_amd64.ln: $(BIGNUMDIR)/amd64/bignum_amd64.c
        @($(LHEAD) $(LINT.c) $(BIGNUMDIR)/amd64/bignum_amd64.c $(LTAIL))

$(LINTS_DIR)/bignum_amd64_asm.ln: $(BIGNUMDIR)/amd64/bignum_amd64_asm.s
        @($(LHEAD) $(LINT.s) $(BIGNUMDIR)/amd64/bignum_amd64_asm.s $(LTAIL))
```