
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

```

1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 # Common Development and Distribution License, Version 1.0 only
8 # (the "License"). You may not use this file except in compliance
9 # with the License.
10 #
11 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
12 # or http://www.opensolaris.org/os/licensing.
13 # See the License for the specific language governing permissions
14 # and limitations under the License.
15 #
16 # When distributing Covered Code, include this CDDL HEADER in each
17 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
18 # If applicable, add the following below this CDDL HEADER, with the
19 # fields enclosed by brackets "[]" replaced with your own identifying
20 # information: Portions Copyright [yyyy] [name of copyright owner]
21 #
22 # CDDL HEADER END
23 #
24 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
25 # Copyright 2005 Sun Microsystems, Inc. All rights reserved.
26 # Use is subject to license terms.
27 #
28 #ident "%Z%M% %I% %E% SMI"
29
30 A note on the file mont_mul.c.
31 A note on the file mont_mul.c. It is used as a starting point
32 for the following hand optimized assembly files.
33
34 It is used as a starting point for the following hand optimized assembly files:
35
36 - usr/src/common/bignum/sun4u/mont_mul_v8plus.s
37 This file is used in the 32-bit pkcs11_softtoken library.
38
39 - usr/src/common/bignum/sun4u/mont_mul_v9.s
40 This file is used in the 64-bit pkcs11_softtoken library.
41
42 - usr/src/common/bignum/sun4u/mont_mul_kernel_v9.s
43 - usr/src/uts/sun4u/rsa/mont_mul.c
44 This file is used in the kernel RSA module.
45
46 We keep the .c file around for future reference.
47
48 See file bignumimpl.c for information about these preprocessor symbols:
49 USE_FLOATING_POINT, PSR_MUL, HWCAP, UMUL64
50 UMUL64 is set in bignum.h based on architecture.
51 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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  * Common Development and Distribution License, Version 1.0 only
8  * (the "License"). You may not use this file except in compliance
9  * with the License.
10 *
11 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
12 * or http://www.opensolaris.org/os/licensing.
13 * See the License for the specific language governing permissions
14 * and limitations under the License.
15 *
16 * When distributing Covered Code, include this CDDL HEADER in each
17 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
18 * If applicable, add the following below this CDDL HEADER, with the
19 * fields enclosed by brackets "[]" replaced with your own identifying
20 * information: Portions Copyright [yyyy] [name of copyright owner]
21 *
22 * CDDL HEADER END
23 */
24 /*
25  * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
26  * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
27  * Use is subject to license terms.
28 */
29 #pragma ident "%Z%%M% %I% %E% SMI"
30
31 /*
32  * This file contains bignum implementation code that
33  * is specific to AMD64, but which is still more appropriate
34  * to write in C, rather than assembly language.
35  * bignum_amd64_asm.s does all the assembly language code
36  * for AMD64 specific bignum support. The assembly language
37  * source file has pure code, no data. Let the C compiler
38  * generate what is needed to handle the variations in
39  * data representation and addressing, for example,
40  * statically linked vs PIC.
41 */
42
43 #include "bignum.h"
44
45 /*
46  * The bignum interface deals with arrays of 64-bit "chunks" or "digits".
47  * Data should be aligned on 8-byte address boundaries for best performance.
48  * The bignum interface deals only with arrays of 32-bit "digits".
49  * The 64-bit bignum functions are internal implementation details.
50  * If a bignum happens to be aligned on a 64-bit boundary
51  * and its length is even, then the pure 64-bit implementation
52  * can be used.
53 */
54
55 #define ISALIGNED64(p) (((uintptr_t)(p) & 7) == 0)
56 #define ISBIGNUM64(p, len) (ISALIGNED64(p) && ((len) & 1) == 0)
57
58 #if defined(__lint)

```

```

59 extern uint64_t *P64(uint32_t *addr);
60 #else /* lint */
61 #define P64(addr) ((uint64_t *)addr)
62 #endif /* lint */
63
64 extern uint64_t big_mul_set_vec64(uint64_t *, uint64_t *, int, uint64_t);
65 extern uint64_t big_mul_add_vec64(uint64_t *, uint64_t *, int, uint64_t);
66 extern void big_mul_vec64(uint64_t *, uint64_t *, int, uint64_t *, int);
67 extern void big_sqr_vec64(uint64_t *, uint64_t *, int);
68
69 extern uint32_t big_mul_set_vec32(uint32_t *, uint32_t *, int, uint32_t);
70 extern uint32_t big_mul_add_vec32(uint32_t *, uint32_t *, int, uint32_t);
71 extern void big_mul_vec32(uint32_t *, uint32_t *, int, uint32_t *, int);
72 extern void big_sqr_vec32(uint32_t *, uint32_t *, int);
73
74 uint32_t big_mul_set_vec(uint32_t *, uint32_t *, int, uint32_t);
75 uint32_t big_mul_add_vec(uint32_t *, uint32_t *, int, uint32_t);
76 void big_mul_vec(uint32_t *, uint32_t *, int, uint32_t *, int);
77 void big_sqr_vec(uint32_t *, uint32_t *, int);
78
79 void
80 big_mul_vec(BIG_CHUNK_TYPE *r, BIG_CHUNK_TYPE *a, int alen,
81             BIG_CHUNK_TYPE *b, int blen)
82 {
83     big_mul_vec(uint32_t *r, uint32_t *a, int alen, uint32_t *b, int blen)
84     {
85         if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen) || !ISBIGNUM64(b, blen)) {
86             big_mul_vec32(r, a, alen, b, blen);
87             return;
88         }
89         big_mul_vec64(P64(r), P64(a), alen / 2, P64(b), blen / 2);
90     }
91 }
92
93 void
94 big_sqr_vec(uint32_t *r, uint32_t *a, int alen)
95 {
96     if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen)) {
97         big_mul_vec32(r, a, alen, a, alen);
98         return;
99     }
100     big_sqr_vec64(P64(r), P64(a), alen / 2);
101 }
102
103 /*
104  * It is OK to cast the 64-bit carry to 32 bit.
105  * There will be no loss, because although we are multiplying the vector, a,
106  * by a uint64_t, its value cannot exceed that of a uint32_t.
107 */
108
109 uint32_t
110 big_mul_set_vec(uint32_t *r, uint32_t *a, int alen, uint32_t digit)
111 {
112     if (!ISALIGNED64(r) || !ISBIGNUM64(a, alen))
113         return (big_mul_set_vec32(r, a, alen, digit));
114     return (big_mul_set_vec64(P64(r), P64(a), alen / 2, digit));
115 }
116
117 uint32_t
118 big_mul_add_vec(uint32_t *r, uint32_t *a, int alen, uint32_t digit)
119 {
120     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 {
128     int i;

129     r[alen] = big_mul_set_vec(r, a, alen, b[0]);
130     r[alen] = big_mul_set_vec64(r, a, alen, b[0]);
131     for (i = 1; i < blen; ++i)
132         r[alen + i] = big_mul_add_vec(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;

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

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;

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

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;

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

```

```

183         --alen;
184     }
185     return ((uint32_t)cy);
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

unchanged_portion_omitted_

new/usr/src/common/bignum/amd64/bignum_amd64_asm.s

1

```
*****
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
*****

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  * Common Development and Distribution License, Version 1.0 only
8  * (the "License"). You may not use this file except in compliance
9  * with the License.
10 *
11 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
12 * or http://www.opensolaris.org/os/licensing.
13 * See the License for the specific language governing permissions
14 * and limitations under the License.
15 *
16 * When distributing Covered Code, include this CDDL HEADER in each
17 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
18 * If applicable, add the following below this CDDL HEADER, with the
19 * fields enclosed by brackets "[]" replaced with your own identifying
20 * information: Portions Copyright [yyyy] [name of copyright owner]
21 *
22 * CDDL HEADER END
23 */
24
25 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
26 * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
27 * Use is subject to license terms.
28
29 #pragma ident "%Z%%M% %I% %E% SMI"
30
31 #include <sys/asm_linkage.h>
32
33 #if defined(lint) || defined(__lint)
34
35 #include <sys/types.h>
36
37 /* ARGSUSED */
38 uint64_t
39 big_mul_set_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
40 { return (0); }
41
42 /* ARGSUSED */
43 uint64_t
44 big_mul_add_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
45 { return (0); }
46
47 /* ARGSUSED */
48 void
49 big_sqr_vec(uint64_t *r, uint64_t *a, int len)
50 { }
51
52 #else /* lint */
53
54 / -----
55 /
```

new/usr/src/common/bignum/amd64/bignum_amd64_asm.s

2

```
51 / Implementation of big_mul_set_vec which exploits
52 / the 64x64->128 bit unsigned multiply instruction.
53 /
54 / As defined in Sun's bignum library for pkcs11, bignums are
55 / composed of an array of 64-bit "digits" or "chunks" along with
56 / descriptive information.
57 / composed of an array of 32-bit "digits" along with descriptive
58 / information. The arrays of digits are only required to be
59 / aligned on 32-bit boundary. This implementation works only
60 / when the two factors and the result happen to be 64 bit aligned
61 / and have an even number of digits.
62 / -----
63
64 / r = a * digit, r and a are vectors of length len
65 / returns the carry digit
66 / r and a are 64 bit aligned.
67 /
68 / uint64_t
69 / big_mul_set_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)
70 / big_mul_set_vec64(uint64_t *r, uint64_t *a, int len, uint64_t digit)
71 /
72 ENTRY(big_mul_set_vec)
73 ENTRY(big_mul_set_vec64)
74 xorq %rax, %rax / if (len == 0) return (0)
75 testq %rdx, %rdx
76 jz .L17
77
78 movq %rdx, %r8 / Use r8 for len; %rdx is used by mul
79 xorq %r9, %r9 / cy = 0
80
81 .L15:
82 cmpq $8, %r8 / 8 - len
83 jb .L16
84 movq 0(%rsi), %rax / rax = a[0]
85 movq 8(%rsi), %r11 / prefetch a[1]
86 mulq %rcx / p = a[0] * digit
87 addq %r9, %rax
88 adcq $0, %rdx / p += cy
89 movq %rax, 0(%rdi) / r[0] = lo(p)
90 movq %rdx, %r9 / cy = hi(p)
91
92 movq %r11, %rax / prefetch a[2]
93 movq 16(%rsi), %r11 / p = a[1] * digit
94 mulq %rcx
95 addq %r9, %rax
96 adcq $0, %rdx / p += cy
97 movq %rax, 8(%rdi) / r[1] = lo(p)
98 movq %rdx, %r9 / cy = hi(p)
99
100 movq %r11, %rax / prefetch a[3]
101 movq 24(%rsi), %r11 / p = a[2] * digit
102 mulq %rcx
103 addq %r9, %rax
104 adcq $0, %rdx / p += cy
105 movq %rax, 16(%rdi) / r[2] = lo(p)
106 movq %rdx, %r9 / cy = hi(p)
107
108 movq %r11, %rax / prefetch a[4]
109 movq 32(%rsi), %r11 / p = a[3] * digit
110 mulq %rcx
111 addq %r9, %rax
112 adcq $0, %rdx / p += cy
113 movq %rax, 24(%rdi) / r[3] = lo(p)
114 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

.L16:
148      movq    0(%rsi), %rax
149      mulq    %rcx                / p = a[0] * digit
150      addq    %r9, %rax
151      adcq    $0, %rdx            / p += cy
152      movq    %rax, 0(%rdi)      / r[0] = lo(p)
153      movq    %rdx, %r9          / cy = hi(p)
154      decq    %r8
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.
227 / composed of an array of 32-bit "digits" along with descriptive
228 / information. The arrays of digits are only required to be
229 / aligned on 32-bit boundary. This implementation works only
230 / when the two factors and the result happen to be 64 bit aligned
231 / and have an even number of digits.
232 / -----
233 /
234 / r += a * digit, r and a are vectors of length len
235 / returns the carry digit
236 / r and a are 64 bit aligned.
237 /
238 / uint64_t
239 / big_mul_add_vec(uint64_t *r, uint64_t *a, int len, uint64_t digit)

```

```

243 / big_mul_add_vec64(uint64_t *r, uint64_t *a, int len, uint64_t digit)
236 /
237 ENTRY(big_mul_add_vec)
245 ENTRY(big_mul_add_vec64)
238 xorq    %rax, %rax          / if (len == 0) return (0)
239 testq   %rdx, %rdx
240 jz      .L27

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

245 .L25:
246 cmpq    $8, %r8           / 8 - len
247 jb      .L26
248 movq    0(%rsi), %rax      / rax = a[0]
249 movq    0(%rdi), %r10     / r10 = r[0]
250 movq    8(%rsi), %r11     / prefetch a[1]
251 mulq    %rcx              / p = a[0] * digit
252 addq    %r10, %rax
253 adcq    $0, %rdx           / p += r[0]
254 movq    8(%rdi), %r10     / prefetch r[1]
255 addq    %r9, %rax
256 adcq    $0, %rdx           / p += cy
257 movq    %rax, 0(%rdi)     / r[0] = lo(p)
258 movq    %rdx, %r9        / cy = hi(p)

260 movq    %r11, %rax
261 movq    16(%rsi), %r11    / prefetch a[2]
262 mulq    %rcx              / p = a[1] * digit
263 addq    %r10, %rax
264 adcq    $0, %rdx           / p += r[1]
265 movq    16(%rdi), %r10    / prefetch r[2]
266 addq    %r9, %rax
267 adcq    $0, %rdx           / p += cy
268 movq    %rax, 8(%rdi)     / r[1] = lo(p)
269 movq    %rdx, %r9        / cy = hi(p)

271 movq    %r11, %rax
272 movq    24(%rsi), %r11    / prefetch a[3]
273 mulq    %rcx              / p = a[2] * digit
274 addq    %r10, %rax
275 adcq    $0, %rdx           / p += r[2]
276 movq    24(%rdi), %r10    / prefetch r[3]
277 addq    %r9, %rax
278 adcq    $0, %rdx           / p += cy
279 movq    %rax, 16(%rdi)    / r[2] = lo(p)
280 movq    %rdx, %r9        / cy = hi(p)

282 movq    %r11, %rax
283 movq    32(%rsi), %r11    / prefetch a[4]
284 mulq    %rcx              / p = a[3] * digit
285 addq    %r10, %rax
286 adcq    $0, %rdx           / p += r[3]
287 movq    32(%rdi), %r10    / prefetch r[4]
288 addq    %r9, %rax
289 adcq    $0, %rdx           / p += cy
290 movq    %rax, 24(%rdi)    / r[3] = lo(p)
291 movq    %rdx, %r9        / cy = hi(p)

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

```

```

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

304 movq    %r11, %rax
305 movq    48(%rsi), %r11    / prefetch a[6]
306 mulq    %rcx              / p = a[5] * digit
307 addq    %r10, %rax
308 adcq    $0, %rdx           / p += r[5]
309 movq    48(%rdi), %r10    / prefetch r[6]
310 addq    %r9, %rax
311 adcq    $0, %rdx           / p += cy
312 movq    %rax, 40(%rdi)    / r[5] = lo(p)
313 movq    %rdx, %r9        / cy = hi(p)

315 movq    %r11, %rax
316 movq    56(%rsi), %r11    / prefetch a[7]
317 mulq    %rcx              / p = a[6] * digit
318 addq    %r10, %rax
319 adcq    $0, %rdx           / p += r[6]
320 movq    56(%rdi), %r10    / prefetch r[7]
321 addq    %r9, %rax
322 adcq    $0, %rdx           / p += cy
323 movq    %rax, 48(%rdi)    / r[6] = lo(p)
324 movq    %rdx, %r9        / cy = hi(p)

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

335 addq    $64, %rsi
336 addq    $64, %rdi
337 subq    $8, %r8

339 jz      .L27
340 jmp     .L25

342 .L26:
343 movq    0(%rsi), %rax
344 movq    0(%rdi), %r10
345 mulq    %rcx              / p = a[0] * digit
346 addq    %r10, %rax
347 adcq    $0, %rdx           / p += r[0]
348 addq    %r9, %rax
349 adcq    $0, %rdx           / p += cy
350 movq    %rax, 0(%rdi)     / r[0] = lo(p)
351 movq    %rdx, %r9        / cy = hi(p)
352 decq    %r8
353 jz      .L27

355 movq    8(%rsi), %rax
356 movq    8(%rdi), %r10
357 mulq    %rcx              / p = a[1] * digit
358 addq    %r10, %rax
359 adcq    $0, %rdx           / p += r[1]
360 addq    %r9, %rax
361 adcq    $0, %rdx           / p += cy
362 movq    %rax, 8(%rdi)     / r[1] = lo(p)
363 movq    %rdx, %r9        / cy = hi(p)
364 decq    %r8
365 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)

447     ENTRY(big_sqr_vec)
448     ENTRY(big_sqr_vec64)
449     pushq   %rbx
450     pushq   %rbp
451     pushq   %r12
452     pushq   %r13
453     pushq   %r14
454     pushq   %r15
455     pushq   %rdx                / save arg3, len
456     pushq   %rsi                / save arg2, a
457     pushq   %rdi                / save arg1, r

458     leaq    8(%rdi), %r13       / tr = r + 1
459     movq    %rsi, %r14         / ta = a
460     movq    %rdx, %r15         / tlen = len
461     decq    %r15               / tlen = len - 1
462     movq    %r13, %rdi         / arg1 = tr
463     leaq    8(%r14), %rsi      / arg2 = ta + 1
464     movq    %r15, %rdx         / arg3 = tlen
465     movq    0(%r14), %rcx      / arg4 = ta[0]
466     call    big_mul_set_vec
467     call    big_mul_set_vec64
468     movq    %rax, 0(%r13, %r15, 8) / tr[tlen] = cy
469     .L31:
470     decq    %r15               / --tlen
471     jz      .L32               / while (--tlen != 0)

472     addq    $16, %r13          / tr += 2
473     addq    $8, %r14            / ++ta
474     movq    %r13, %rdi         / arg1 = tr
475     leaq    8(%r14), %rsi      / arg2 = ta + 1
476     movq    %r15, %rdx         / arg3 = tlen
477     movq    0(%r14), %rcx      / arg4 = ta[0]
478     call    big_mul_add_vec
479     call    big_mul_add_vec64
480     movq    %rax, 0(%r13, %r15, 8) / tr[tlen] = cy
481     jmp     .L31

482     .L32:

483 / No more function calls after this.
484 / Restore arguments to registers.
485 / However, don't use %rdx for arg3, len, because it is heavily
486 / used by the hardware MUL instruction. Use %r8, instead.
487     movq    0(%rsp), %rdi      / %rdi == arg1 == r
488     movq    8(%rsp), %rsi      / %rsi == arg2 == a
489     movq    16(%rsp), %r8      / %r8 == arg3 == len

490     movq    0(%rsi), %rax      / %rax = a[0];
491     mulq    %rax               / s = %edx:%eax = a[0]**2
492     movq    %rax, 0(%rdi)      / r[0] = lo64(s)
493     movq    %rdx, %r9          / cy = hi64(s)
494     xorq    %rdx, %rdx
495     movq    8(%rdi), %rax      / p = %rdx:%rax = r[1]
496     addq    %rax, %rax
497     adcq    $0, %rdx           / p = p << 1
498     addq    %r9, %rax
499     adcq    $0, %rdx           / p = (r[1] << 1) + cy
500     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 */

```

new/usr/src/common/bignum/bignum.h

1

```
*****
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */
26 #ifndef _BIGNUM_H
27 #define _BIGNUM_H
28
29 #pragma ident "%Z%%M% %I% %E% SMI"
30
31 #ifdef __cplusplus
32 extern "C" {
33 #endif
34
35 #include <sys/types.h>
36
37 #if defined(__sparcv9) || defined(__amd64) /* 64-bit chunk size */
38 #define BIGNUM_CHUNK_32
39 #else
40 #define UMUL64 /* 64-bit multiplication results are supported */
41 #endif
42
43 #define BITSINBYTE 8
44
45 /* Bignum "digits" (aka "chunks" or "words") are either 32- or 64-bits */
46 #ifdef BIGNUM_CHUNK_32
47 #define BIG_CHUNK_SIZE 32
48 #define BIG_CHUNK_TYPE uint32_t
49 #define BIG_CHUNK_TYPE_SIGNED int32_t
50 #define BIG_CHUNK_HIGHBIT 0x80000000
51 #define BIG_CHUNK_ALLBITS 0xffffffff
52 #endif
```

new/usr/src/common/bignum/bignum.h

2

```
53 #define BIG_CHUNK_LOWHALFBITS 0xffff
54 #define BIG_CHUNK_HALF_HIGHBIT 0x8000
55
56 #else
57 #define BIG_CHUNK_SIZE 64
58 #define BIG_CHUNK_TYPE uint64_t
59 #define BIG_CHUNK_TYPE_SIGNED int64_t
60 #define BIG_CHUNK_HIGHBIT 0x8000000000000000ULL
61 #define BIG_CHUNK_ALLBITS 0xffffffffffffffffULL
62 #define BIG_CHUNK_LOWHALFBITS 0xffffffffFULL
63 #define BIG_CHUNK_HALF_HIGHBIT 0x80000000ULL
64 #endif
65
66 #define BITLEN2BIGNUMLEN(x) (((x) + BIG_CHUNK_SIZE - 1) / BIG_CHUNK_SIZE)
67 #define CHARLEN2BIGNUMLEN(x) (((x) + sizeof (BIG_CHUNK_TYPE) - 1) / \
68     sizeof (BIG_CHUNK_TYPE))
69
70 #define BIGNUM_WORDSIZE (BIG_CHUNK_SIZE / BITSINBYTE) /* word size in bytes */
71 #define BIG_CHUNKS_FOR_160BITS ((160 + BIG_CHUNK_SIZE - 1) / BIG_CHUNK_SIZE)
72
73 /*
74 * leading 0's are permitted
75 * 0 should be represented by size>=1, size>=len>=1, sign=1,
76 * value[i]=0 for 0<i<len
77 */
78 typedef struct {
79     /* size and len in units of BIG_CHUNK_TYPE words */
80     uint32_t size; /* size of memory allocated for value */
81     uint32_t len; /* number of valid data words in value */
82     int size; /* size of memory allocated for value */
83     int len; /* number of words that hold valid data in value */
84     int sign; /* 1 for nonnegative, -1 for negative */
85     int malloced; /* 1 if value was malloced, 0 if not */
86     BIG_CHUNK_TYPE *value;
87 } BIGNUM;
88
89 #endif
90
91 #endif
92
93 #endif
94
95 #endif
96
97 #endif
98
99 #endif
100
101 #endif
102
103 #endif
104
105 #endif
106
107 #endif
108
109 #endif
110
111 #endif
112
113 #endif
114
115 #endif
116
117 #endif
118
119 #endif
120
121 #endif
122
123 #endif
124
125 #endif
126
127 #endif
128
129 #endif
130
131 #endif
132
133 #endif
134
135 #endif
136
137 #endif
138
139 #endif
140
141 #endif
142
143 #endif
144
145 #endif
146
147 #endif
148
149 #endif
150
151 #endif
152
153 #endif
154
155 #endif
156
157 #endif
158
159 #endif
160
161 #endif
162
163 #endif
164
165 #endif
166
167 #endif
168
169 #endif
170
171 #endif
172
173 #endif
174
175 #endif
176
177 #endif
178
179 #endif
180
181 #endif
182
183 #endif
184
185 #endif
186
187 #endif
188
189 #endif
190
191 #endif
192
193 #endif
194
195 #endif
196
197 #endif
198
199 #endif
200
201 #endif
202
203 #endif
204
205 #endif
206
207 #endif
208
209 #endif
210
211 #endif
212
213 #endif
214
215 #endif
216
217 #endif
218
219 #endif
220
221 #endif
222
223 #endif
224
225 #endif
226
227 #endif
228
229 #endif
230
231 #endif
232
233 #endif
234
235 #endif
236
237 #endif
238
239 #endif
240
241 #endif
242
243 #endif
244
245 #endif
246
247 #endif
248
249 #endif
250
251 #endif
252
253 #endif
254
255 #endif
256
257 #endif
258
259 #endif
260
261 #endif
262
263 #endif
264
265 #endif
266
267 #endif
268
269 #endif
270
271 #endif
272
273 #endif
274
275 #endif
276
277 #endif
278
279 #endif
280
281 #endif
282
283 #endif
284
285 #endif
286
287 #endif
288
289 #endif
290
291 #endif
292
293 #endif
294
295 #endif
296
297 #endif
298
299 #endif
300
301 #endif
302
303 #endif
304
305 #endif
306
307 #endif
308
309 #endif
310
311 #endif
312
313 #endif
314
315 #endif
316
317 #endif
318
319 #endif
320
321 #endif
322
323 #endif
324
325 #endif
326
327 #endif
328
329 #endif
330
331 #endif
332
333 #endif
334
335 #endif
336
337 #endif
338
339 #endif
340
341 #endif
342
343 #endif
344
345 #endif
346
347 #endif
348
349 #endif
350
351 #endif
352
353 #endif
354
355 #endif
356
357 #endif
358
359 #endif
360
361 #endif
362
363 #endif
364
365 #endif
366
367 #endif
368
369 #endif
370
371 #endif
372
373 #endif
374
375 #endif
376
377 #endif
378
379 #endif
380
381 #endif
382
383 #endif
384
385 #endif
386
387 #endif
388
389 #endif
390
391 #endif
392
393 #endif
394
395 #endif
396
397 #endif
398
399 #endif
400
401 #endif
402
403 #endif
404
405 #endif
406
407 #endif
408
409 #endif
410
411 #endif
412
413 #endif
414
415 #endif
416
417 #endif
418
419 #endif
420
421 #endif
422
423 #endif
424
425 #endif
426
427 #endif
428
429 #endif
430
431 #endif
432
433 #endif
434
435 #endif
436
437 #endif
438
439 #endif
440
441 #endif
442
443 #endif
444
445 #endif
446
447 #endif
448
449 #endif
450
451 #endif
452
453 #endif
454
455 #endif
456
457 #endif
458
459 #endif
460
461 #endif
462
463 #endif
464
465 #endif
466
467 #endif
468
469 #endif
470
471 #endif
472
473 #endif
474
475 #endif
476
477 #endif
478
479 #endif
480
481 #endif
482
483 #endif
484
485 #endif
486
487 #endif
488
489 #endif
490
491 #endif
492
493 #endif
494
495 #endif
496
497 #endif
498
499 #endif
500
501 #endif
502
503 #endif
504
505 #endif
506
507 #endif
508
509 #endif
510
511 #endif
512
513 #endif
514
515 #endif
516
517 #endif
518
519 #endif
520
521 #endif
522
523 #endif
524
525 #endif
526
527 #endif
528
529 #endif
530
531 #endif
532
533 #endif
534
535 #endif
536
537 #endif
538
539 #endif
540
541 #endif
542
543 #endif
544
545 #endif
546
547 #endif
548
549 #endif
550
551 #endif
552
553 #endif
554
555 #endif
556
557 #endif
558
559 #endif
560
561 #endif
562
563 #endif
564
565 #endif
566
567 #endif
568
569 #endif
570
571 #endif
572
573 #endif
574
575 #endif
576
577 #endif
578
579 #endif
580
581 #endif
582
583 #endif
584
585 #endif
586
587 #endif
588
589 #endif
590
591 #endif
592
593 #endif
594
595 #endif
596
597 #endif
598
599 #endif
600
601 #endif
602
603 #endif
604
605 #endif
606
607 #endif
608
609 #endif
610
611 #endif
612
613 #endif
614
615 #endif
616
617 #endif
618
619 #endif
620
621 #endif
622
623 #endif
624
625 #endif
626
627 #endif
628
629 #endif
630
631 #endif
632
633 #endif
634
635 #endif
636
637 #endif
638
639 #endif
640
641 #endif
642
643 #endif
644
645 #endif
646
647 #endif
648
649 #endif
650
651 #endif
652
653 #endif
654
655 #endif
656
657 #endif
658
659 #endif
660
661 #endif
662
663 #endif
664
665 #endif
666
667 #endif
668
669 #endif
670
671 #endif
672
673 #endif
674
675 #endif
676
677 #endif
678
679 #endif
680
681 #endif
682
683 #endif
684
685 #endif
686
687 #endif
688
689 #endif
690
691 #endif
692
693 #endif
694
695 #endif
696
697 #endif
698
699 #endif
700
701 #endif
702
703 #endif
704
705 #endif
706
707 #endif
708
709 #endif
710
711 #endif
712
713 #endif
714
715 #endif
716
717 #endif
718
719 #endif
720
721 #endif
722
723 #endif
724
725 #endif
726
727 #endif
728
729 #endif
730
731 #endif
732
733 #endif
734
735 #endif
736
737 #endif
738
739 #endif
740
741 #endif
742
743 #endif
744
745 #endif
746
747 #endif
748
749 #endif
750
751 #endif
752
753 #endif
754
755 #endif
756
757 #endif
758
759 #endif
760
761 #endif
762
763 #endif
764
765 #endif
766
767 #endif
768
769 #endif
770
771 #endif
772
773 #endif
774
775 #endif
776
777 #endif
778
779 #endif
780
781 #endif
782
783 #endif
784
785 #endif
786
787 #endif
788
789 #endif
790
791 #endif
792
793 #endif
794
795 #endif
796
797 #endif
798
799 #endif
800
801 #endif
802
803 #endif
804
805 #endif
806
807 #endif
808
809 #endif
810
811 #endif
812
813 #endif
814
815 #endif
816
817 #endif
818
819 #endif
820
821 #endif
822
823 #endif
824
825 #endif
826
827 #endif
828
829 #endif
830
831 #endif
832
833 #endif
834
835 #endif
836
837 #endif
838
839 #endif
840
841 #endif
842
843 #endif
844
845 #endif
846
847 #endif
848
849 #endif
850
851 #endif
852
853 #endif
854
855 #endif
856
857 #endif
858
859 #endif
860
861 #endif
862
863 #endif
864
865 #endif
866
867 #endif
868
869 #endif
870
871 #endif
872
873 #endif
874
875 #endif
876
877 #endif
878
879 #endif
880
881 #endif
882
883 #endif
884
885 #endif
886
887 #endif
888
889 #endif
890
891 #endif
892
893 #endif
894
895 #endif
896
897 #endif
898
899 #endif
900
901 #endif
902
903 #endif
904
905 #endif
906
907 #endif
908
909 #endif
910
911 #endif
912
913 #endif
914
915 #endif
916
917 #endif
918
919 #endif
920
921 #endif
922
923 #endif
924
925 #endif
926
927 #endif
928
929 #endif
930
931 #endif
932
933 #endif
934
935 #endif
936
937 #endif
938
939 #endif
940
941 #endif
942
943 #endif
944
945 #endif
946
947 #endif
948
949 #endif
950
951 #endif
952
953 #endif
954
955 #endif
956
957 #endif
958
959 #endif
960
961 #endif
962
963 #endif
964
965 #endif
966
967 #endif
968
969 #endif
970
971 #endif
972
973 #endif
974
975 #endif
976
977 #endif
978
979 #endif
980
981 #endif
982
983 #endif
984
985 #endif
986
987 #endif
988
989 #endif
990
991 #endif
992
993 #endif
994
995 #endif
996
997 #endif
998
999 #endif
1000
1001 #endif
1002
1003 #endif
1004
1005 #endif
1006
1007 #endif
1008
1009 #endif
1010
1011 #endif
1012
1013 #endif
1014
1015 #endif
1016
1017 #endif
1018
1019 #endif
1020
1021 #endif
1022
1023 #endif
1024
1025 #endif
1026
1027 #endif
1028
1029 #endif
1030
1031 #endif
1032
1033 #endif
1034
1035 #endif
1036
1037 #endif
1038
1039 #endif
1040
1041 #endif
1042
1043 #endif
1044
1045 #endif
1046
1047 #endif
1048
1049 #endif
1050
1051 #endif
1052
1053 #endif
1054
1055 #endif
1056
1057 #endif
1058
1059 #endif
1060
1061 #endif
1062
1063 #endif
1064
1065 #endif
1066
1067 #endif
1068
1069 #endif
1070
1071 #endif
1072
1073 #endif
1074
1075 #endif
1076
1077 #endif
1078
1079 #endif
1080
1081 #endif
1082
1083 #endif
1084
1085 #endif
1086
1087 #endif
1088
1089 #endif
1090
1091 #endif
1092
1093 #endif
1094
1095 #endif
1096
1097 #endif
1098
1099 #endif
1100
1101 #endif
1102
1103 #endif
1104
1105 #endif
1106
1107 #endif
1108
1109 #endif
1110
1111 #endif
1112
1113 #endif
1114
1115 #endif
1116
1117 #endif
1118
1119 #endif
1120
1121 #endif
1122
1123 #endif
1124
1125 #endif
1126
1127 #endif
1128
1129 #endif
1130
1131 #endif
1132
1133 #endif
1134
1135 #endif
1136
1137 #endif
1138
1139 #endif
1140
1141 #endif
1142
1143 #endif
1144
1145 #endif
1146
1147 #endif
1148
1149 #endif
1150
1151 #endif
1152
1153 #endif
1154
1155 #endif
1156
1157 #endif
1158
1159 #endif
1160
1161 #endif
1162
1163 #endif
1164
1165 #endif
1166
1167 #endif
1168
1169 #endif
1170
1171 #endif
1172
1173 #endif
1174
1175 #endif
1176
1177 #endif
1178
1179 #endif
1180
1181 #endif
1182
1183 #endif
1184
1185 #endif
1186
1187 #endif
1188
1189 #endif
1190
1191 #endif
1192
1193 #endif
1194
1195 #endif
1196
1197 #endif
1198
1199 #endif
1200
1201 #endif
1202
1203 #endif
1204
1205 #endif
1206
1207 #endif
1208
1209 #endif
1210
1211 #endif
1212
1213 #endif
1214
1215 #endif
1216
1217 #endif
1218
1219 #endif
1220
1221 #endif
1222
1223 #endif
1224
1225 #endif
1226
1227 #endif
1228
1229 #endif
1230
1231 #endif
1232
1233 #endif
1234
1235 #endif
1236
1237 #endif
1238
1239 #endif
1240
1241 #endif
1242
1243 #endif
1244
1245 #endif
1246
1247 #endif
1248
1249 #endif
1250
1251 #endif
1252
1253 #endif
1254
1255 #endif
1256
1257 #endif
1258
1259 #endif
1260
1261 #endif
1262
1263 #endif
1264
1265 #endif
1266
1267 #endif
1268
1269 #endif
1270
1271 #endif
1272
1273 #endif
1274
1275 #endif
1276
1277 #endif
1278
1279 #endif
1280
1281 #endif
1282
1283 #endif
1284
1285 #endif
1286
1287 #endif
1288
1289 #endif
1290
1291 #endif
1292
1293 #endif
1294
1295 #endif
1296
1297 #endif
1298
1299 #endif
1300
1301 #endif
1302
1303 #endif
1304
1305 #endif
1306
1307 #endif
1308
1309 #endif
1310
1311 #endif
1312
1313 #endif
1314
1315 #endif
1316
1317 #endif
1318
1319 #endif
1320
1321 #endif
1322
1323 #endif
1324
1325 #endif
1326
1327 #endif
1328
1329 #endif
1330
1331 #endif
1332
1333 #endif
1334
1335 #endif
1336
1337 #endif
1338
1339 #endif
1340
1341 #endif
1342
1343 #endif
1344
1345 #endif
1346
1347 #endif
1348
1349 #endif
1350
1351 #endif
1352
1353 #endif
1354
1355 #endif
1356
1357 #endif
1358
1359 #endif
1360
1361 #endif
1362
1363 #endif
1364
1365 #endif
1366
1367 #endif
1368
1369 #endif
1370
1371 #endif
1372
1373 #endif
1374
1375 #endif
1376
1377 #endif
1378
1379 #endif
1380
1381 #endif
1382
1383 #endif
1384
1385 #endif
1386
1387 #endif
1388
1389 #endif
1390
1391 #endif
1392
1393 #endif
1394
1395 #endif
1396
1397 #endif
1398
1399 #endif
1400
1401 #endif
1402
1403 #endif
1404
1405 #endif
1406
1407 #endif
1408
1409 #endif
1410
1411 #endif
1412
1413 #endif
1414
1415 #endif
1416
1417 #endif
1418
1419 #endif
1420
1421 #endif
1422
1423 #endif
1424
1425 #endif
1426
1427 #endif
1428
1429 #endif
1430
1431 #endif
1432
1433 #endif
1434
1435 #endif
1436
1437 #endif
1438
1439 #endif
1440
1441 #endif
1442
1443 #endif
1444
1445 #endif
1446
1447 #endif
1448
1449 #endif
1450
1451 #endif
1452
1453 #endif
1454
1455 #endif
1456
1457 #endif
1458
1459 #endif
1460
1461 #endif
1462
1463 #endif
1464
1465 #endif
1466
1467 #endif
1468
1469 #endif
1470
1471 #endif
1472
1473 #endif
1474
1475 #endif
1476
1477 #endif
1478
1479 #endif
1480
1481 #endif
1482
1483 #endif
1484
1485 #endif
1486
1487 #endif
1488
1489 #endif
1490
1491 #endif
1492
1493 #endif
1494
1495 #endif
1496
1497 #endif
1498
1499 #endif
1500
1501 #endif
1502
1503 #endif
1504
1505 #endif
1506
1507 #endif
1508
1509 #endif
1510
1511 #endif
1512
1513 #endif
1514
1515 #endif
1516
1517 #endif
1518
1519 #endif
1520
1521 #endif
1522
1523 #endif
1524
1525 #endif
1526
1527 #endif
1528
1529 #endif
1530
1531 #endif
1532
1533 #endif
1534
1535 #endif
1536
1537 #endif
1538
1539 #endif
1540
1541 #endif
1542
1543 #endif
1544
1545 #endif
1546
1547 #endif
1548
1549 #endif
1550
1551 #endif
1552
1553 #endif
1554
1555 #endif
1556
1557 #endif
1558
1559 #endif
1560
1561 #endif
1562
1563 #endif
1564
1565 #endif
1566
1567 #endif
1568
1569 #endif
1570
1571 #endif
1572
1573 #endif
1574
1575 #endif
1576
1577 #endif
1578
1579 #endif
1580
1581 #endif
1582
1583 #endif
1584
1585 #endif
1586
1587 #endif
1588
1589 #endif
1590
1591 #endif
1592
1593 #endif
1594
1595 #endif
1596
1597 #endif
1598
1599 #endif
1600
1601 #endif
1602
1603 #endif
1604
1605 #endif
1606
1607 #endif
1608
1609 #endif
1610
1611 #endif
1612
1613 #endif
1614
1615 #endif
1616
1617 #endif
1618
1619 #endif
1620
1621 #endif
1622
1623 #endif
1624
1625 #endif
1626
1627 #endif
1628
1629 #endif
1630
1631 #endif
1632
1633 #endif
1634
1635 #endif
1636
1637 #endif
1638
1639 #endif
1640
1641 #endif
1642
1643 #endif
1644
1645 #endif
1646
1647 #endif
1648
1649 #endif
1650
1651 #endif
1652
1653 #endif
1654
1655 #endif
1656
1657 #endif
1658
1659 #endif
1660
1661 #endif
1662
1663 #endif
1664
1665 #endif
1666
1667 #endif
1668
1669 #endif
1670
1671 #endif
1672
1673 #endif
1674
1675 #endif
1676
1677 #endif
1678
1679 #endif
1680
1681 #endif
1682
1683 #endif
1684
1685 #endif
1686
1687 #endif
1688
1689 #endif
1690
1691 #endif
1692
1693 #endif
1694
1695 #endif
1696
1697 #endif
1698
1699 #endif
1700
1701 #endif
1702
1703 #endif
1704
1705 #endif
1706
1707 #endif
1708
1709 #endif
1710
1711 #endif
1712
1713 #endif
1714
1715 #endif
1716
1717 #endif
1718
1719 #endif
1720
1721 #endif
1722
1723 #endif
1724
1725 #endif
1726
1727 #endif
1728
1729 #endif
1730
1731 #endif
1732
1733 #endif
1734
1735 #endif
1736
1737 #endif
1738
1739 #endif
1740
1741 #endif
1742
1743 #endif
1744
1745 #endif
1746
1747 #endif
1748
1749 #endif
1750
1751 #endif
1752
1753 #endif
1754
1755 #endif
1756
1757 #endif
1758
1759 #endif
1760
1761 #endif
1762
1763 #endif
1764
1765 #endif
1766
1767 #endif
1768
1769 #endif
1770
1771 #endif
1772
1773 #endif
1774
1775 #endif
1776
1777 #endif
1778
1779 #endif
1780
1781 #endif
1782
1783 #endif
1784
1785 #endif
1786
1787 #endif
1788
1789 #endif
1790
1791 #endif
1792
1793 #endif
1794
1795 #endif
1796
1797 #endif
1798
1799 #endif
1800
1801 #endif
1802
1803 #endif
1804
1805 #endif
1806
1807 #endif
1808
1809 #endif
1810
1811 #endif
1812
1813 #endif
1814
1815 #endif
1816
1817 #endif
1818
1819 #endif
1820
1821 #endif
1822
1823 #endif
1824
1825 #endif
1826
1827 #endif
1828
1829 #endif
1830
1831 #endif
1832
1833 #endif
1834
1835 #endif
1836
1837 #endif
1838
1839 #endif
1840
1841 #endif
1842
1843 #endif
1844
1845 #endif
1846
1847 #endif
1848
1849 #endif
1850
1851 #endif
1852
1853 #endif
1854
1855 #endif
1856
1857 #endif
1858
1859 #endif
1860
1861 #endif
1862
1863 #endif
1864
1865 #endif
1866
1867 #endif
1868
1869 #endif
1870
1871 #endif
1872
1873 #endif
1874
1875 #endif
1876
1877 #endif
1878
1879 #endif
1880
1881 #endif
1882
1883 #endif
1884
1885 #endif
1886
1887 #endif
1888
1889 #endif
1890
1891 #endif
1892
1893 #endif
1894
1895 #endif
1896
1897 #endif
1898
1899 #endif
1900
1901 #endif
1902
1903 #endif
1904
1905 #endif
1906
1907 #endif
1908
1909 #endif
1910
1911 #endif
1912
1913 #endif
1914
1915 #endif
1916
1917 #endif
1918
1919 #endif
1920
1921 #endif
1922
1923 #endif
1924
1925 #endif
1926
1927 #endif
1928
1929 #endif
1930
1931 #endif
1932
1933 #endif
1934
1935 #endif
1936
1937 #endif
1938
1939 #endif
1940
1941 #endif
1942
1943 #endif
1944
1945 #endif
1946
1947 #endif
1948
1949 #endif
1950
1951 #endif
1952
1953 #endif
1954
1955 #endif
1956
1957 #endif
1958
1959 #endif
1960
1961 #endif
1962
1963 #endif
1964
1965 #endif
1966
1967 #endif
1968
1969 #endif
1970
1971 #endif
1972
1973 #endif
1974
1975 #endif
1976
1977 #endif
1978
1979 #endif
1980
1981 #endif
1982
1983 #endif
1984
1985 #endif
1986
1987 #endif
1988
1989 #endif
1990
1991 #endif
1992
1993 #endif
1994
1995 #endif
1996
1997 #endif
1998
1999 #endif
2000
2001 #endif
2002
2003 #endif
2004
2005 #endif
2006
2007 #endif
2008
2009 #endif
2010
2011 #endif
2012
2013 #endif
2014
2015 #endif
2016
2017 #endif
2018
2019 #endif
2020
2021 #endif
2022
2023 #endif
2024
2025 #endif
2026
2027 #endif
2028
2029 #endif
2030
2031 #endif
2032
2033 #endif
2034
2035 #endif
2036
2037 #endif
2038
2039 #endif
2040
2041 #endif
2042
2043 #endif
2044
2045 #endif
2046
2047 #endif
2048
2049 #endif
2050
2051 #endif
2052
2053 #endif
2054
2055 #endif
2056
2057 #endif
2058
2059 #endif
2060
2061 #endif
2062
2063 #endif
2064
2065 #endif
2066
2067 #endif
2068
2069 #endif
2070
2071 #endif
2072
2073
```

new/usr/src/common/bignum/bignumimpl.c

1

```
*****
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24 * Use is subject to license terms.
25 */
26 #pragma ident "%Z%M% %I% %E% SMI"
27
28 #define big_div_pos_fast big_div_pos
29
30 #include "bignum.h"
31
32 /*
33 * Configuration guide
34 * -----
35 *
36 * There are 4 preprocessor symbols used to configure the bignum
37 * implementation. This file contains no logic to configure based on
38 * processor; we leave that to the Makefiles to specify.
39 *
40 * USE_FLOATING_POINT
41 * Meaning: There is support for a fast floating-point implementation of
42 * Montgomery multiply.
43 *
44 * PSR_MUL
45 * Meaning: There are processor-specific versions of the low level
46 * functions to implement big_mul. Those functions are: big_mul_set_vec,
47 * big_mul_add_vec, big_mul_vec, and big_sqr_vec. PSR_MUL implies support
48 * for all 4 functions. You cannot pick and choose which subset of these
49 * functions to support; that would lead to a rat's nest of #ifdefs.
50 *
51 * HWCAP
52 * Meaning: Call multiply support functions through a function pointer.
53 * On x86, there are multiple implementations for different hardware
54 * capabilities, such as MMX, SSE2, etc. Tests are made at run-time, when
55 * a function is first used. So, the support functions are called through
56 * a function pointer. There is no need for that on Sparc, because there
57 * is only one implementation; support functions are called directly.
```

new/usr/src/common/bignum/bignumimpl.c

2

```
52 * Later, if there were some new VIS instruction, or something, and a
53 * run-time test were needed, rather than variant kernel modules and
54 * libraries, then HWCAP would be defined for Sparc, as well.
55 *
56 * UMUL64
57 * Meaning: It is safe to use generic C code that assumes the existence
58 * of a 32 x 32 --> 64 bit unsigned multiply. If this is not defined,
59 * then the generic code for big_mul_add_vec() must necessarily be very slow,
60 * because it must fall back to using 16 x 16 --> 32 bit multiplication.
61 *
62 */
63
64 #include <sys/types.h>
65 #include "bignum.h"
66
67 #ifdef _KERNEL
68 #include <sys/ddi.h>
69 #include <sys/mdesc.h>
70 #include <sys/crypto/common.h>
71
72 #include <sys/types.h>
73 #include <sys/kmem.h>
74 #include <sys/param.h>
75 #include <sys/sunddi.h>
76
77 #else
78 #include <stdlib.h>
79 #include <stdio.h>
80 #include <assert.h>
81 #define ASSERT assert
82 #endif /* _KERNEL */
83
84 #ifdef _LP64 /* truncate 64-bit size_t to 32-bits */
85 #define UI32(ui) ((uint32_t)ui)
86 #else /* size_t already 32-bits */
87 #define UI32(ui) (ui)
88 #endif
89
90 #ifdef _KERNEL
91 #define big_malloc(size) kmem_alloc(size, KM_NOSLEEP)
92 #define big_free(ptr, size) kmem_free(ptr, size)
93
94 void *
95 big_realloc(void *from, size_t oldsize, size_t newsize)
96 {
97     void *rv;
98
99     rv = kmem_alloc(newsize, KM_NOSLEEP);
100     if (rv != NULL)
101         bcopy(from, rv, oldsize);
102     kmem_free(from, oldsize);
103     return (rv);
104 }
105
106 #else /* _KERNEL */
107
108 #include <stdlib.h>
109 #include <stdio.h>
110 #include <assert.h>
111 #define ASSERT assert
112
113 #ifndef MALLOC_DEBUG
114 #define big_malloc(size) malloc(size)
115 #endif
```

new/usr/src/common/bignum/bignumimpl.c

3

```
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 }
122 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;
144
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)) {
150             if ((i % 8 == 0) && (i != 0)) {
151                 (void) printf("\n");
152             }
153 #else
154         (void) printf("%08x %08x ", (uint32_t)((a->value[i] >> 32),
155             (uint32_t)((a->value[i] & 0xffffffff)));
156         if (((i & 3) == 0) && (i != 0)) { /* end of this chunk */
157             if ((i % 4 == 0) && (i != 0)) {
158                 (void) printf("\n");
159             }
160 #endif
161         }
162     }
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  */
183 /* size in BIG_CHUNK_SIZE-bit words */
```

new/usr/src/common/bignum/bignumimpl.c

4

```
183 BIG_ERR_CODE
184 big_init(BIGNUM *number, int size)
185 {
186     number->value = big_malloc(BIGNUM_WORDSIZE * size);
187     number->value = big_malloc(sizeof (BIG_CHUNK_TYPE) * size);
188     if (number->value == NULL) {
189         return (BIG_NO_MEM);
190     }
191     number->size = size;
192     number->len = 0;
193     number->sign = 1;
194     number->malloced = 1;
195     return (BIG_OK);
196 }
```

```
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  */
218 /* size in BIG_CHUNK_SIZE-bit words */
219 BIG_ERR_CODE
220 big_init1(BIGNUM *number, int size, BIG_CHUNK_TYPE *buf, int bufsize)
221 {
222     if ((buf == NULL) || (size > bufsize)) {
223         number->value = big_malloc(BIGNUM_WORDSIZE * size);
224         number->value = big_malloc(sizeof (BIG_CHUNK_TYPE) * size);
225         if (number->value == NULL) {
226             return (BIG_NO_MEM);
227         }
228         number->size = size;
229         number->malloced = 1;
230     } else {
231         number->value = buf;
232         number->size = bufsize;
233         number->malloced = 0;
234     }
235     number->len = 0;
236     number->sign = 1;
237     return (BIG_OK);
238 }
```

```
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
257  * (len + sizeof (BIG_CHUNK_TYPE) - 1) / sizeof (BIG_CHUNK_TYPE) bytes
258  * converts from byte-big-endian format to bignum format (words in
259  * little endian order, but bytes within the words big endian)
260 */
261 void
262 bytestring2bignum(BIGNUM *bn, uchar_t *kn, size_t len)
263 {
264     int i, j;
265     uint32_t offs;
266     const uint32_t slen = UI32(len);
267     int i, j, offs;
268     BIG_CHUNK_TYPE word;
269     uchar_t *knwordp;

270     if (slen == 0) {
271         bn->len = 1;
272         bn->value[0] = 0;
273         return;
274     }
275 #ifdef _LP64
276     offs = (uint32_t)len % sizeof (BIG_CHUNK_TYPE);
277     bn->len = (uint32_t)len / sizeof (BIG_CHUNK_TYPE);

278     offs = slen % BIGNUM_WORDSIZE;
279     bn->len = slen / BIGNUM_WORDSIZE;

280     for (i = 0; i < slen / BIGNUM_WORDSIZE; i++) {
281         knwordp = &(kn[slen - BIGNUM_WORDSIZE * (i + 1)]);
282         for (j = 1; j < (uint32_t)len / sizeof (BIG_CHUNK_TYPE); j++) {
283             /* !_LP64 */
284             offs = len % sizeof (BIG_CHUNK_TYPE);
285             bn->len = len / sizeof (BIG_CHUNK_TYPE);
286             for (i = 0; i < len / sizeof (BIG_CHUNK_TYPE); i++) {
287                 /* !_LP64 */
288                 knwordp = &(kn[len - sizeof (BIG_CHUNK_TYPE) * (i + 1)]);
289                 word = knwordp[0];
290                 for (j = 1; j < BIGNUM_WORDSIZE; j++) {
291                     word = (word << BITSINBYTE) + knwordp[j];
292                     for (j = 1; j < sizeof (BIG_CHUNK_TYPE); j++) {
293                         word = (word << 8) + knwordp[j];
294                     }
295                     bn->value[i] = word;
296                 }
297             }
298             if (offs > 0) {
299                 word = kn[0];
300                 for (i = 1; i < offs; i++) word = (word << BITSINBYTE) + kn[i];
301                 for (i = 1; i < offs; i++) word = (word << 8) + kn[i];
302                 bn->value[bn->len++] = word;
303             }
304             while ((bn->len > 1) && (bn->value[bn->len - 1] == 0)) {
305                 while ((bn->len > 1) && (bn->value[bn->len - 1] == 0)) {
306                     bn->len--;
307                 }
308             }
309         }
310     }

```

```

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                         kn[j < sizeof (BIG_CHUNK_TYPE); j++] {
263                             kn[slen - 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[slen / 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[slen - sizeof (BIG_CHUNK_TYPE) * i - j - 1] =

```

```

333         word & 0xff;
334         word = word >> BITSINBYTE;
335         word = word >> 8;
336     }
337     for (i = 0; i < slen - BIGNUM_WORDSIZE * bn->len; i++) {
338 #ifdef _LP64
339         for (i = 0;
340              i < (uint32_t)len - sizeof (BIG_CHUNK_TYPE) * bn->len;
341              i++) {
342 #else /* !_LP64 */
343         for (i = 0; i < len - sizeof (BIG_CHUNK_TYPE) * bn->len; i++) {
344 #endif /* !_LP64 */
345             kn[i] = 0;
346         }
347     }
348 }
349 }
350 }

```

```

351 int
352 big_bitlength(BIGNUM *a)
353 {
354     int l = 0, b = 0;
355     BIG_CHUNK_TYPE c;
356
357     l = a->len - 1;
358     while ((l > 0) && (a->value[l] == 0)) {
359         l--;
360     }
361     b = BIG_CHUNK_SIZE;
362     b = sizeof (BIG_CHUNK_TYPE) * BITSINBYTE;
363     c = a->value[l];
364     while ((b > 1) && ((c & BIG_CHUNK_HIGHBIT) == 0)) {
365         c = c << 1;
366         b--;
367     }
368
369     return (1 * BIG_CHUNK_SIZE + b);
370     return (1 * sizeof (BIG_CHUNK_TYPE) * BITSINBYTE + b);
371 }

```

```

372 BIG_ERR_CODE
373 big_copy(BIGNUM *dest, BIGNUM *src)
374 {
375     BIG_CHUNK_TYPE *newptr;
376     int i, len;
377
378     len = src->len;
379     while ((len > 1) && (src->value[len - 1] == 0)) {
380         len--;
381     }
382     src->len = len;
383     if (dest->size < len) {
384         if (dest->malloced == 1) {
385             newptr = (BIG_CHUNK_TYPE *)big_realloc(dest->value,
386             BIGNUM_WORDSIZE * dest->size,
387             BIGNUM_WORDSIZE * len);
388             sizeof (BIG_CHUNK_TYPE) * dest->size,
389             sizeof (BIG_CHUNK_TYPE) * len);
390         } else {
391             newptr = (BIG_CHUNK_TYPE *)
392             big_malloc(BIGNUM_WORDSIZE * len);
393             big_malloc(sizeof (BIG_CHUNK_TYPE) * len);
394             if (newptr != NULL) {
395                 dest->malloced = 1;

```

```

386     }
387     }
388     if (newptr == NULL) {
389         return (BIG_NO_MEM);
390     }
391     dest->value = newptr;
392     dest->size = len;
393     dest->len = len;
394     dest->sign = src->sign;
395     for (i = 0; i < len; i++) {
396         dest->value[i] = src->value[i];
397     }
398
399     return (BIG_OK);
400 }
401 }

```

```

402 BIG_ERR_CODE
403 big_extend(BIGNUM *number, int size)
404 {
405     BIG_CHUNK_TYPE *newptr;
406     int i;
407
408     if (number->size >= size)
409         return (BIG_OK);
410     if (number->malloced) {
411         number->value = big_realloc(number->value,
412         BIGNUM_WORDSIZE * number->size,
413         BIGNUM_WORDSIZE * size);
414         sizeof (BIG_CHUNK_TYPE) * number->size,
415         sizeof (BIG_CHUNK_TYPE) * size);
416     } else {
417         newptr = big_malloc(BIGNUM_WORDSIZE * size);
418         newptr = big_malloc(sizeof (BIG_CHUNK_TYPE) * size);
419         if (newptr != NULL) {
420             for (i = 0; i < number->size; i++) {
421                 newptr[i] = number->value[i];
422             }
423             number->value = newptr;
424         }
425
426         if (number->value == NULL) {
427             return (BIG_NO_MEM);
428         }
429
430         number->size = size;
431         number->malloced = 1;
432         return (BIG_OK);
433 }

```

unchanged_portion_omitted

```

434 /* returns -1 if |aa|<|bb|, 0 if |aa|==|bb| 1 if |aa|>|bb| */
435 int
436 big_cmp_abs(BIGNUM *aa, BIGNUM *bb)
437 {
438     int i;
439
440     if (aa->len > bb->len) {
441         for (i = aa->len - 1; i > bb->len - 1; i--) {
442             if (aa->value[i] > 0) {
443                 return (1);
444             }
445         }
446     }
447 }

```

```

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;
601         i = aa->len-1;
602     }
603     for (; i >= 0; i--) {
604         if (aa->value[i] > bb->value[i]) {
605             return (1);
606         } else if (aa->value[i] < bb->value[i]) {
607             return (-1);
608         }
609     }
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;
941
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;
952         result->value[i-1] = (ai << (BIG_CHUNK_SIZE - offs)) | cy;
953         cy = ai >> offs;
954     }
955     result->len = aa->len;
956     result->value[result->len - 1] = cy;
957     result->sign = aa->sign;
958 }

```

```

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)
966 big_div_pos_fast(BIGNUM *result, BIGNUM *remainder, BIGNUM *aa, BIGNUM *bb)
967 {
968     BIG_ERR_CODE err = BIG_OK;
969     int i, alen, blen, tlen, rlen, offs;
970     BIG_CHUNK_TYPE higha, highb, coeff;
971     BIG_CHUNK_TYPE *a, *b;
972     BIGNUM bbhigh, bblow, tresult, tmp1, tmp2;
973     BIG_CHUNK_TYPE tmp1value[BIGTMP_SIZE];
974     BIG_CHUNK_TYPE tmp2value[BIGTMP_SIZE];
975     BIG_CHUNK_TYPE tresultvalue[BIGTMP_SIZE];
976     BIG_CHUNK_TYPE bblowvalue[BIGTMP_SIZE];

```

```

976     BIG_CHUNK_TYPE bbhighvalue[BIGTMP_SIZE];
977
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     }
993
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     }
1006
1007     if ((err = big_init1(&bblow, blen + 1,
1008         bblowvalue, arraysize(bblowvalue))) != BIG_OK)
1009         return (err);
1010
1011     if ((err = big_init1(&bbhigh, blen + 1,
1012         bbhighvalue, arraysize(bbhighvalue))) != BIG_OK)
1013         goto ret1;
1014
1015     if ((err = big_init1(&tmp1, alen + 2,
1016         tmp1value, arraysize(tmp1value))) != BIG_OK)
1017         goto ret2;
1018
1019     if ((err = big_init1(&tmp2, blen + 2,
1020         tmp2value, arraysize(tmp2value))) != BIG_OK)
1021         goto ret3;
1022
1023     if ((err = big_init1(&tresult, alen - blen + 2,
1024         tresultvalue, arraysize(tresultvalue))) != BIG_OK)
1025         goto ret4;
1026
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     }
1037
1038     big_shiftleft(&bblow, bb, offs);
1039
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]; \
1147     pf = (uint64_t)a[R+1]; \
1148     t = p + cy; \
1149     r[R] = (uint32_t)t; \
1150     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]; \
1161     pf = (uint64_t)a[R+1]; \
1162     t = p + t + cy; \
1163     r[R] = (uint32_t)t; \
1164     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 }
1227 #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;
1318         p = ((uint64_t)r[col + 1] << 1) + cy;
1319         r[col + 1] = (uint32_t)p;
1320         cy = p >> 32;
1321         ++row;
1322         col += 2;
1323     }
1324     r[col + 1] = (uint32_t)cy;
1325     r[col + 1] = (uint32_t)cy;
1326 }
1327 #endif /* UNROLL8 */
1329 void
1330 big_sqr_vec(BIG_CHUNK_TYPE *r, BIG_CHUNK_TYPE *a, int len)
1331 {
1332     int i;
1334     ASSERT(r != a);
1335     r[len] = big_mul_set_vec(r, a, len, a[0]);
1336     for (i = 1; i < len; ++i)
1337         r[i] = big_mul_add_vec(r + i, a, len, a[i]);
1338     r[i] = big_mul_add_vec(r + i, a, len, a[i]);
1339 }
1340 #endif /* BIG_CHUNK_SIZE == 32/64 */
1342 #else /* ! UMUL64 */
1343 #if (BIG_CHUNK_SIZE != 32)
1344 #error Don't use 64-bit chunks without defining UMUL64
1345 #endif
1347 /*
1348  * r = r + a * digit, r and a are vectors of length len
1349  * returns the carry digit

```

```

1417 */
1418 uint32_t
1419 big_mul_add_vec(uint32_t *r, uint32_t *a, int len, uint32_t digit)
1420 {
1421     uint32_t cy, cy1, retcy, dlow, dhigh;
1422     int i;

1424     cy1 = 0;
1425     dlow = digit & 0xffff;
1426     dhigh = digit >> 16;
1427     for (i = 0; i < len; i++) {
1428         cy = (cy1 >> 16) + dlow * (a[i] & 0xffff) + (r[i] & 0xffff);
1429         cy1 = (cy >> 16) + dlow * (a[i] >> 16) + (r[i] >> 16);
1430         r[i] = (cy & 0xffff) | (cy1 << 16);
1431     }
1432     retcy = cy1 >> 16;

1434     cy1 = r[0] & 0xffff;
1435     for (i = 0; i < len - 1; i++) {
1436         cy = (cy1 >> 16) + dhigh * (a[i] & 0xffff) + (r[i] >> 16);
1437         r[i] = (cy1 & 0xffff) | (cy << 16);
1438         cy1 = (cy >> 16) + dhigh * (a[i] >> 16) + (r[i + 1] & 0xffff);
1439     }
1440     cy = (cy1 >> 16) + dhigh * (a[len - 1] & 0xffff) + (r[len - 1] >> 16);
1441     r[len - 1] = (cy1 & 0xffff) | (cy << 16);
1442     retcy = (cy >> 16) + dhigh * (a[len - 1] >> 16) + retcy;

1444     return (retcy);
1445 }

    unchanged_portion_omitted

1465 void
1466 big_sqr_vec(uint32_t *r, uint32_t *a, int len)
1467 {
1468     int i;

1470     ASSERT(r != a);
1471     r[len] = big_mul_set_vec(r, a, len, a[0]);
1472     for (i = 1; i < len; ++i)
1473         r[len + i] = big_mul_add_vec(r + i, a, len, a[i]);
1435     r[len + i] = big_mul_add_vec(r + i, a, len, a[i]);
1474 }

1476 #endif /* UMUL64 */

1478 void
1479 big_mul_vec(BIG_CHUNK_TYPE *r, BIG_CHUNK_TYPE *a, int alen,
1480             BIG_CHUNK_TYPE *b, int blen)
1481 {
1482     int i;

1484     r[alen] = big_mul_set_vec(r, a, alen, b[0]);
1485     for (i = 1; i < blen; ++i)
1486         r[alen + i] = big_mul_add_vec(r + i, a, alen, b[i]);
1448     r[alen + i] = big_mul_add_vec(r + i, a, alen, b[i]);
1487 }

1490 #endif /* ! PSR_MUL */

1493 /*
1494  * result = aa * bb  result->value should be big enough to hold the result
1495  *
1496  * Implementation: Standard grammar school algorithm
1497  */

```

```

1498 */
1499 BIG_ERR_CODE
1500 big_mul(BIGNUM *result, BIGNUM *aa, BIGNUM *bb)
1501 {
1502     BIGNUM tmp1;
1503     BIG_CHUNK_TYPE tmp1value[BIGTMP_SIZE];
1504     BIG_CHUNK_TYPE *r, *t, *a, *b;
1505     BIG_ERR_CODE err;
1506     int i, alen, blen, rsize, sign, diff;

1508     if (aa == bb) {
1509         diff = 0;
1510     } else {
1511         diff = big_cmp_abs(aa, bb);
1512         if (diff < 0) {
1513             BIGNUM *tt;
1514             tt = aa;
1515             aa = bb;
1516             bb = tt;
1517         }
1518     }
1519     a = aa->value;
1520     b = bb->value;
1521     alen = aa->len;
1522     blen = bb->len;
1523     while ((alen > 1) && (a[alen - 1] == 0)) {
1524         alen--;
1525     }
1526     aa->len = alen;
1527     while ((blen > 1) && (b[blen - 1] == 0)) {
1528         blen--;
1529     }
1530     bb->len = blen;

1532     rsize = alen + blen;
1533     ASSERT(rsize > 0);
1534     if (result->size < rsize) {
1535         err = big_extend(result, rsize);
1536         if (err != BIG_OK) {
1537             return (err);
1538         }
1539     }
1540     /* aa or bb might be an alias to result */
1541     a = aa->value;
1542     b = bb->value;
1543     r = result->value;

1545     if (((alen == 1) && (a[0] == 0)) || ((blen == 1) && (b[0] == 0))) {
1546         result->len = 1;
1547         result->sign = 1;
1548         r[0] = 0;
1549         return (BIG_OK);
1550     }
1551     sign = aa->sign * bb->sign;
1552     if ((alen == 1) && (a[0] == 1)) {
1553         for (i = 0; i < blen; i++) {
1554             r[i] = b[i];
1555         }
1556         result->len = blen;
1557         result->sign = sign;
1558         return (BIG_OK);
1559     }
1560     if ((blen == 1) && (b[0] == 1)) {
1561         for (i = 0; i < alen; i++) {
1562             r[i] = a[i];
1563         }
1564     }

```

```

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);
1599     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, needssubtract;
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     needssubtract = 0;
1642     if ((rr[2 * nlen] != 0))
1643         needssubtract = 1;
1644     else {
1645         for (i = 2 * nlen - 1; i >= nlen; i--) {
1646             if (rr[i] > nn[i - nlen]) {
1647                 needssubtract = 1;
1648                 break;
1649             } else if (rr[i] < nn[i - nlen]) {
1650                 break;
1651             }
1652         }
1653     }
1654     if (needssubtract)
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--)
1664         ;
1665     ret->len = i + 1;
1666     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[BIGTMP_SIZE];

```

```

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

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

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

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

1832     for (i = 0; i < apowersize; 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 < apowersize; i++) {
1847         if ((err = big_mont_mul(&tmp1, ma,
1848             &apowers[i - 1], n, n0)) != BIG_OK) {
1849             &apowers[i - 1], n, n0)) != BIG_OK) {
1850                 goto ret;
1851             }
1852         (void) big_copy(&apowers[i], &tmp1);
1853     }

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 + 1 + 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 = apowersize - 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;
1959     int          i, j, k, l, m, p, bit, bitind, bitcount, nlen;
1960     double       dn0;
1961     double       *dn, *dt, *d16r, *d32r;
1962     uint32_t     *nint, *prod;
1963     double       *apowers[APOWERS_MAX_SIZE];
1964     uint32_t     nbits, groupbits, apowerssize;
1965     int          nbits, groupbits, apowerssize;
1966     BIG_ERR_CODE err = BIG_OK;
1968     #ifdef _KERNEL
1969     uint8_t fpu[sizeof(kfpu_t) + FPR_ALIGN];
1970     kfpu_t *fpu;
1972     #ifdef DEBUG
1973     if (!fpu_exists)
1974         return (BIG_GENERAL_ERR);
1975     #endif
1977     fpu = (kfpu_t *)P2ROUNDUP((uintptr_t)fpu, FPR_ALIGN);
1978     big_savefp(fpu);
1980     nlen = 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] & 0xffffffffFULL);
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] & 0xffffffffFULL);
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 < apowersize; 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->nlen; 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->nlen; i++) {
2083         prod[2 * i] = (uint32_t)(tmp->value[i] & 0xffffffffFULL);
2084         prod[2 * i + 1] = (uint32_t)(tmp->value[i] >> 32);
2085     }
2086     for (i = tmp->nlen * 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 + 1 + 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 >> (1 + 1)],
2124                         apowers[p >> (1 + 1)],
2125                         dt, dn, nint, nlen, dn0);
2126                     for (m = 0; m < l; m++) {
2127                         conv_i32_to_d32_and_d16(d32r,
2128                             d16r, prod, nlen);
2129                         mont_mulf_noconv(prod, d32r,
2130                             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 >> (1 + 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->nlen = i + 1;

2172 ret:
2173     for (i = apowersize - 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     }

```

```

2196 #ifdef _KERNEL
2197     big_restorefp(fpu);
2198 #endif
2200     return (err);
2201 }
2203 #endif /* USE_FLOATING_POINT */

2206 BIG_ERR_CODE
2207 big_modexp_ext(BIGNUM *result, BIGNUM *a, BIGNUM *e, BIGNUM *n, BIGNUM *n_rr,
2208     big_modexp_ncp_info_t *info)
2209 {
2210     BIGNUM      ma, tmp, rr;
2211     BIG_CHUNK_TYPE mvalue[BIGTMP_SIZE];
2212     BIG_CHUNK_TYPE tmpvalue[BIGTMP_SIZE];
2213     BIG_CHUNK_TYPE rrvalue[BIGTMP_SIZE];
2214     BIG_ERR_CODE err;
2215     BIG_CHUNK_TYPE n0;

2217     if ((err = big_init1(&ma, n->len, mvalue, arraysize(mvalue))) !=
2218         BIG_OK) {
2219         return (err);
2220     }
2221     ma.len = 1;
2222     ma.value[0] = 0;

2224     if ((err = big_init1(&tmp, 2 * n->len + 1,
2225         tmpvalue, arraysize(tmpvalue))) != BIG_OK) {
2226         goto ret1;
2227     }

2229     /* clear the malloced bit to help cleanup */
2230     /* set the malloced bit to help cleanup */
2231     rr.malloced = 0;

2232     if (n_rr == NULL) {
2233         if ((err = big_init1(&rr, 2 * n->len + 1,
2234             rrvalue, arraysize(rrvalue))) != BIG_OK) {
2235             goto ret2;
2236         }
2237         if (big_mont_rr(&rr, n) != BIG_OK) {
2238             goto ret;
2239         }
2240         n_rr = &rr;
2241     }

2243     n0 = big_n0(n->value[0]);

2245     if (big_cmp_abs(a, n) > 0) {
2246         if ((err = big_div_pos(NULL, &ma, a, n)) != BIG_OK) {
2247             goto ret;
2248         }
2249         err = big_mont_conv(&ma, &ma, n, n0, n_rr);
2250     } else {
2251         err = big_mont_conv(&ma, a, n, n0, n_rr);
2252     }
2253     if (err != BIG_OK) {
2254         goto ret;
2255     }

2257     tmp.len = 1;
2258     tmp.value[0] = 1;
2259     if ((err = big_mont_conv(&tmp, &tmp, n, n0, n_rr)) != BIG_OK) {
2260         goto ret;

```

```

2261     }

2263     if ((info != NULL) && (info->func != NULL)) {
2264         err = (*(info->func))(&tmp, &ma, e, n, &tmp, n0,
2265             info->ncp, info->reqp);
2266     } else {
2267         err = big_modexp_ncp_sw(&tmp, &ma, e, n, &tmp, n0);
2268     }
2269     if (err != BIG_OK) {
2270         goto ret;
2271     }

2273     ma.value[0] = 1;
2274     ma.len = 1;
2275     if ((err = big_mont_mul(&tmp, &tmp, &ma, n, n0)) != BIG_OK) {
2276         goto ret;
2277     }
2278     err = big_copy(result, &tmp);

2280 ret:
2281     if (rr.malloced) {
2282         big_finish(&rr);
2283     }
2284 ret2:
2285     big_finish(&tmp);
2286 ret1:
2287     big_finish(&ma);

2289     return (err);
2290 }
    unchanged_portion_omitted

2409 static BIG_CHUNK_TYPE onearr[1] = {(BIG_CHUNK_TYPE)1};
2410 BIGNUM big_One = {1, 1, 1, 0, onearr};

2412 static BIG_CHUNK_TYPE twoarr[1] = {(BIG_CHUNK_TYPE)2};
2413 BIGNUM big_Two = {1, 1, 1, 0, twoarr};

2415 static BIG_CHUNK_TYPE fourarr[1] = {(BIG_CHUNK_TYPE)4};
2416 static BIGNUM big_Four = {1, 1, 1, 0, fourarr};

2419 BIG_ERR_CODE
2420 big_sqrt_pos(BIGNUM *result, BIGNUM *n)
2421 {
2422     BIGNUM      *high, *low, *mid, *t;
2423     BIGNUM      t1, t2, t3, prod;
2424     BIG_CHUNK_TYPE t1value[BIGTMP_SIZE];
2425     BIG_CHUNK_TYPE t2value[BIGTMP_SIZE];
2426     BIG_CHUNK_TYPE t3value[BIGTMP_SIZE];
2427     BIG_CHUNK_TYPE prodvalue[BIGTMP_SIZE];
2428     int          i, diff;
2429     uint32_t      nbits, nrootbits, highbits;
2430     int           i, nbits, diff, nrootbits, highbits;
2431     BIG_ERR_CODE err;

2432     nbits = big_numbits(n);

2434     if ((err = big_init1(&t1, n->len + 1,
2435         t1value, arraysize(t1value))) != BIG_OK)
2436         return (err);
2437     if ((err = big_init1(&t2, n->len + 1,
2438         t2value, arraysize(t2value))) != BIG_OK)
2439         goto ret1;
2440     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_HIGHBITS;
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.mallocated) big_finish(&prod);
2500 ret3:
2501 if (t3.mallocated) big_finish(&t3);
2502 ret2:
2503 if (t2.mallocated) big_finish(&t2);
2504 ret1:
2505 if (t1.mallocated) big_finish(&t1);

```

```

2507     return (err);
2508 }
2509 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;
2601     int m, w, i;
2602     BIG_CHUNK_TYPE bit;
2603     BIGNUM ki, tmp, tmp2;
2604     BIG_CHUNK_TYPE kivalue[BIGTMP_SIZE];
2605     BIG_CHUNK_TYPE tmpvalue[BIGTMP_SIZE];
2606     BIG_CHUNK_TYPE tmp2value[BIGTMP_SIZE];
2607     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,
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             }

```

```

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.mallocated) big_finish(&tmp2);
2685 ret2:
2686     if (tmp.mallocated) big_finish(&tmp);
2687 ret1:
2688     if (ki.mallocated) 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_____

```

new/usr/src/common/bignum/mont_mulf.c

1

```
*****
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
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  * Common Development and Distribution License, Version 1.0 only
8  * (the "License"). You may not use this file except in compliance
9  * with the License.
10 *
11 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
12 * or http://www.opensolaris.org/os/licensing.
13 * See the License for the specific language governing permissions
14 * and limitations under the License.
15 *
16 * When distributing Covered Code, include this CDDL HEADER in each
17 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
18 * If applicable, add the following below this CDDL HEADER, with the
19 * fields enclosed by brackets "[]" replaced with your own identifying
20 * information: Portions Copyright [yyyy] [name of copyright owner]
21 *
22 * CDDL HEADER END
23 */
24 /*
25  * Copyright 2009 Sun Microsystems, Inc. All rights reserved.
26  * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
27  * Use is subject to license terms.
28 */
29 #pragma ident "%Z%%M% %I% %E% SMI"
30
31 /*
32  * If compiled without -DRF_INLINE_MACROS then needs -lm at link time
33  * If compiled with -DRF_INLINE_MACROS then needs conv.il at compile time
34  * (i.e. cc <compiler_flags> -DRF_INLINE_MACROS conv.il mont_mulf.c )
35  * (i.e. cc <compiler_flags> -DRF_INLINE_MACROS conv.il mont_mulf.c )
36 */
37
38 #include <sys/types.h>
39 #include <math.h>
40
41 static const double TwoTo16 = 65536.0;
42 static const double TwoToMinus16 = 1.0/65536.0;
43 static const double Zero = 0.0;
44 static const double TwoTo32 = 65536.0 * 65536.0;
45 static const double TwoToMinus32 = 1.0 / (65536.0 * 65536.0);
46
47 #ifdef RF_INLINE_MACROS
48
49 double upper32(double);
50 double lower32(double, double);
51 double mod(double, double, double);
52
53 #else
54
55 static double
56 upper32(double x)
57 {
58     return (floor(x * TwoToMinus32));
59 }
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2
```

```

240 */
241 void mont_mulf_noconv(uint32_t *result,
242                      double *dm1, double *dm2, double *dt,
243                      double *dn, uint32_t *nint,
244                      int nlen, double dn0)
245 {
246     int i, j, jj;
247     double digit, m2j, a, b;
248     double *pdm1, *pdm2, *pdn, *pdtj, pdn_0, pdm1_0;
249
250     pdm1 = &(dm1[0]);
251     pdm2 = &(dm2[0]);
252     pdn = &(dn[0]);
253     pdm2[2 * nlen] = Zero;
254
255     if (nlen != 16) {
256         for (i = 0; i < 4 * nlen + 2; i++)
257             dt[i] = Zero;
258         a = dt[0] = pdm1[0] * pdm2[0];
259         digit = mod(lower32(a, Zero) * dn0, TwoToMinus16, TwoTo16);
260
261         pdtj = &(dt[0]);
262         for (j = 0; j < 2 * nlen; j++, jj++, pdtj++) {
263             m2j = pdm2[jj];
264             a = pdtj[0] + pdn[0] * digit;
265             b = pdtj[1] + pdm1[0] * pdm2[j + 1] + a * TwoToMinus16;
266             pdtj[1] = b;
267
268             #pragma pipeloop(0)
269             for (i = 1; i < nlen; i++) {
270                 pdtj[2 * i] += pdm1[i] * m2j + pdn[i] * digit;
271             }
272             if (jj == 30) {
273                 cleanup(dt, j / 2 + 1, 2 * nlen + 1);
274                 jj = 0;
275             }
276
277             digit = mod(lower32(b, Zero) * dn0,
278                       TwoToMinus16, TwoTo16);
279         }
280     } else {
281         a = dt[0] = pdm1[0] * pdm2[0];
282
283         dt[65] = dt[64] = dt[63] = dt[62] = dt[61] = dt[60] =
284             dt[59] = dt[58] = dt[57] = dt[56] = dt[55] =
285             dt[54] = dt[53] = dt[52] = dt[51] = dt[50] =
286             dt[49] = dt[48] = dt[47] = dt[46] = dt[45] =
287             dt[44] = dt[43] = dt[42] = dt[41] = dt[40] =
288             dt[39] = dt[38] = dt[37] = dt[36] = dt[35] =
289             dt[34] = dt[33] = dt[32] = dt[31] = dt[30] =
290             dt[29] = dt[28] = dt[27] = dt[26] = dt[25] =
291             dt[24] = dt[23] = dt[22] = dt[21] = dt[20] =
292             dt[19] = dt[18] = dt[17] = dt[16] = dt[15] =
293             dt[14] = dt[13] = dt[12] = dt[11] = dt[10] =
294             dt[9] = dt[8] = dt[7] = dt[6] = dt[5] = dt[4] =
295             dt[3] = dt[2] = dt[1] = Zero;
296
297         pdn_0 = pdn[0];
298         pdm1_0 = pdm1[0];
299
300         digit = mod(lower32(a, Zero) * dn0, TwoToMinus16, TwoTo16);
301         pdtj = &(dt[0]);
302
303         for (j = 0; j < 32; j++, pdtj++) {
304             m2j = pdm2[j];

```

```

306         a = pdtj[0] + pdn_0 * digit;
307         b = pdtj[1] + pdm1_0 * pdm2[j + 1] + a * TwoToMinus16;
308         pdtj[1] = b;
309
310         pdtj[2] += pdm1[1] * m2j + pdn[1] * digit;
311         pdtj[4] += pdm1[2] * m2j + pdn[2] * digit;
312         pdtj[6] += pdm1[3] * m2j + pdn[3] * digit;
313         pdtj[8] += pdm1[4] * m2j + pdn[4] * digit;
314         pdtj[10] += pdm1[5] * m2j + pdn[5] * digit;
315         pdtj[12] += pdm1[6] * m2j + pdn[6] * digit;
316         pdtj[14] += pdm1[7] * m2j + pdn[7] * digit;
317         pdtj[16] += pdm1[8] * m2j + pdn[8] * digit;
318         pdtj[18] += pdm1[9] * m2j + pdn[9] * digit;
319         pdtj[20] += pdm1[10] * m2j + pdn[10] * digit;
320         pdtj[22] += pdm1[11] * m2j + pdn[11] * digit;
321         pdtj[24] += pdm1[12] * m2j + pdn[12] * digit;
322         pdtj[26] += pdm1[13] * m2j + pdn[13] * digit;
323         pdtj[28] += pdm1[14] * m2j + pdn[14] * digit;
324         pdtj[30] += pdm1[15] * m2j + pdn[15] * digit;
325         /* no need for cleanup, cannot overflow */
326         /* no need for cleanup, cannot overflow */
327         digit = mod(lower32(b, Zero) * dn0,
328                   TwoToMinus16, TwoTo16);
329     }
330
331     conv_d16_to_i32(result, dt + 2 * nlen, (int64_t *)dt, nlen + 1);
332     adjust_montf_result(result, nint, nlen);
333 }
334
335 unchanged_portion_omitted

```

new/usr/src/lib/pkcs11/libsoftcrypto/amd64/Makefile

1

```

*****
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
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
23 # Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 # lib/pkcs11/libsoftcrypto/amd64/Makefile
26 #

28 LIBRARY=      libsoftcrypto.a
29 VERS= .1

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

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

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

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

44 CFLAGS      += -x04 -xcrossfile
45 BIGNUM_FLAGS += -DPSR_MUL
46 LINTFLAGS64 += $(BIGNUM_FLAGS) $(AES_FLAGS)

47 CLEANFILES += arcfour-x86_64.s

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

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

54 arcfour-x86_64.s:      $(ARCFOUR_DIR)/amd64/arcfour-x86_64.pl
55 $(PERL) $? $@

```

new/usr/src/lib/pkcs11/libsoftcrypto/amd64/Makefile

2

```

57 pics/%.o:      $(AES_DIR)/$(MACH64)/%.c
58 $(COMPILE.c) $(AES_FLAGS) -o $@ $<
59 $(POST_PROCESS_0)

61 pics/%.o:      $(AES_DIR)/$(MACH64)/%.s
62 $(COMPILE.s) $(AES_FLAGS) -o $@ $<
63 $(POST_PROCESS_0)

65 pics/%.o:      $(BIGNUM_DIR)/$(MACH64)/%.c
66 $(COMPILE.c) $(BIGNUM_FLAGS) -o $@ $<
67 $(POST_PROCESS_0)

69 pics/%.o:      $(BIGNUM_DIR)/$(MACH64)/%.s
70 $(COMPILE64.s) $(BIGNUM_FLAGS) -o $@ $<
71 $(POST_PROCESS_0)

73 pics/%.o:      arcfour-x86_64.s
74 $(COMPILE64.s) $(ARCFOUR_FLAGS) -o $@ $<
75 $(POST_PROCESS_0)

```

new/usr/src/uts/intel/bignum/Makefile.64

1

```
*****
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
*****
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 #
22 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
22 # Copyright 2008 Sun Microsystems, Inc. All rights reserved.
23 # Use is subject to license terms.
24 #
25 #ident "%Z%M% %I% %E% SMI"
26 #
27 # Configuration and targets for bignum module
28 # specific to AMD 64-bit architecture, amd64.
29 #
30 # Bignum configuration (BIGNUM_CFG):
31 # PSR_MUL:
32 #   There is a processor-specific implementation bignum multiply functions
33 #   HWCAP:
34 #   There are multiple implementations of bignum functions, and the
35 #   appropriate one must be chosen at run time, based on testing
36 #   hardware capabilities.
37 #
38 # -DPSR_MUL:
39 #   For AMD64, there is a processor-specific implementation of
40 #   the bignum multiply functions, which takes advantage of the
41 #   64x64->128 bit multiply instruction.
42 #
43 # -UHWCAP:
44 #   There is only one implementation, because the 128 bit multiply using
45 #   general-purpose registers is faster than any MMX or SSE2 implementation.
46 #
47 # BIGNUM_CFG = -DPSR_MUL
48 # CFLAGS += -x04 -xcrossfile
49 #
50 $(OBJS_DIR)/bignumimpl.o $(LINTS_DIR)/bignumimpl.ln := \
51   CPPFLAGS += $(BIGNUM_CFG)
52 $(OBJS_DIR)/bignum_amd64.o $(LINTS_DIR)/bignum_amd64.ln := \
53   CPPFLAGS += $(BIGNUM_CFG)
54 $(OBJS_DIR)/bignum_amd64.o: $(BIGNUMDIR)/amd64/bignum_amd64.c
55 $(COMPILE.c) -o $@ $(BIGNUM_CFG) $(BIGNUMDIR)/amd64/bignum_amd64.c
56 $(CTFCONVERT_0)
```

new/usr/src/uts/intel/bignum/Makefile.64

2

```
55 $(POST_PROCESS_0)
56
57 $(OBJS_DIR)/bignum_amd64_asm.o: $(BIGNUMDIR)/amd64/bignum_amd64_asm.s
58 $(COMPILE.s) -P -o $@ $(BIGNUM_CFG) \
59   $(BIGNUMDIR)/amd64/bignum_amd64_asm.s
60 $(COMPILE.s) -P -o $@ $(BIGNUM_CFG) $(BIGNUMDIR)/amd64/bignum_amd64_asm.
60 $(POST_PROCESS_0)
61
62 $(LINTS_DIR)/bignum_amd64.ln: $(BIGNUMDIR)/amd64/bignum_amd64.c
63 @($LHEAD) $(LINT.c) $(BIGNUMDIR)/amd64/bignum_amd64.c $(LTAIL))
64
65 $(LINTS_DIR)/bignum_amd64_asm.ln: $(BIGNUMDIR)/amd64/bignum_amd64_asm.s
66 @($LHEAD) $(LINT.s) $(BIGNUMDIR)/amd64/bignum_amd64_asm.s $(LTAIL))
```