Durham University

Durham E-Theses

Development of crystallographic surfaces for modelling interactions

Ford, Peter S.

How to cite:

Ford, Peter S. (1997) *Development of crystallographic surfaces for modelling interactions*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/4776/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Development of Crystallographic Surfaces for Modelling Interactions
Appendix B

Peter S. Ford

June 13, 1997

The copyright of this thesis rests with the author. No quotation from it should be published without the written consent of the author and information derived from it should be acknowledged.

- 3 JUL 1997

Contents

UNIV DUNELM

Appendix B

Source code for symmetry program

The following section contain the source files for the *Cracker* symmetry program. Header (*.h) files are included at the end.

B.1 Main routine: Cracker

```
/*
 * cracker
 * Extracts the set of spacegroup symmetry operators from which a unit
 * cell can be produced, printing them in string form.
 * This calls the CRACK subroutine to do the hard work.
 *
 * Common Blocks:
 */
#include <crack.h>
/*
 * Functions called:
 */
LOGICAL crack(void);
/*
 * Code:
 */

int main (int argc, char *argv[])
{
   char m[1024];
```

```
    int i;

/*
 * Unscramble command line to get spacegroup
 */
    *spgp.s = NULL;
    *m = NULL;
    for (i=0; i<argc; i++)
    {
        strcat(m,argv[i]);
    }
    for (i=0; m[i] && i<(Sgrpmax - 1); i++)
    {
        spgp.s[i] = m[i];
    }
    spgp.s[i] = NULL;
    verb = True;
    if (!crack())
/*
 * Error decoding spacegroup
 */

    {
        printf("Symbol given : %s\n", spgp.s);
        printf("This spacegroup symbol contains an error\n");
        printf("Please check your input and try again.\n");
        spgp.s[0] = NULL;
    }
    return(0);
}
```

B.2 Crack Subroutine

```
/*
 * crack
 *
 * Spacegroup symbol interpretor designed for use with "builder2"
 * With suitable adjustment this will also stand alone.
 * Within the common block <crack.h> are defined 'symops'; a 193x4x4 double
 * precision array to contain matrices for the symmetry operations generated,
 * and 'nops'; the number of these operations actually present at any stage in
 * the routine.
 *
```

```
 * Common Blocks:
 */
#include <crack.h>
#include <math.h>
/*
 * Functions called:
 */
LOGICAL crunch(void),
invpt(void);
int     selgen(void),
matax(LOGICAL),
symeqs(void);
void    centre(char, int*);
/*
 * Code:
 */
LOGICAL crack(void)
{
   int i ,j ,k, n, ngen;
   LOGICAL ok = True;
/*
 * Send the spacegroup directly to CRUNCH to decode it
 */

   if (!crunch()) return(False);
/*
 * Pass on to matax to get the matrices for the spacegroup: set parameter to
 * True to allow higher order axes to be multiplied up, generating extra
 * operations for 3,4 and 6 fold axes
 */

   if ((nops = matax(True)) == 0) return(False);
/*
 * Do a check for a centre of symmetry
 */

   centric = invpt();
/*
 * Tell the user what we have found out so far
 */
   if (verb)
   {
      printf("Spacegroup %s is %s ", spgp.s, class.s);
      if (centric) printf("Centrosymmetric\n");
      else printf("Non-centrosymmetric\n");
```

```
    }
/*
 * Now select some generators
 */
    ngen = selgen();
/*
 * Make sure the translations on generators are all in range 0 =< t < 1
 */
    for (k=0; k<ngen; k++)
    {
        for (i=0; i<3; i++)
        {
            gmats[k].m[3][i] = gmats[k].m[3][i] % 12;
            if (gmats[k].m[3][i] < 0)
            {
                gmats[k].m[3][i] = 12 - gmats[k].m[3][i];
            }
        }
        gmats[k].m[3][3] = 1;
    }
/*
 * Now expand the generators into a full set of symmetry operators
 */

    nops = symeqs();
/*
 * Put in cell centering if necessary
 */

    switch (centring)
    {
    case 'P':
        break;

    case 'A': case 'B': case 'C': case 'I': case 'R':
        centre(centring, &nops);
        break;
    case 'F':
        centre('A', &nops);
        centre('B', &nops);
        break;

    case 'H':
        centre('H', &nops);
        break;
```

```
    default:
        printf("Unknown centring symbol %c\n",centring);
        ok = False;
        break;
    }
/*
 * Print out the operators
 */

    if (verb)
    {
        if (nops < 188)
        {
            for (i=nops; i<nops+4; i++) *(opstr[i].s) = NULL;
        }
        printf("Operators :\n");
        for (k=0; k<nops; k+=2)
        {
            printf("%30s %30s\n", opstr[k].s,opstr[k+1].s);
        }
    }
/*
 * Copy the integer matrices in to the double precision array
 */

    for (n=0; n<nops; n++)
    {
        for (i=0; i<4; i++)
        {
            for (j=0; j<3; j++)
            {
                symops[n].m[i][j] = (double) isym[n].m[i][j];
            }
            symops[n].m[i][j] = (double) (isym[n].m[i][j]) / 12.0;
        }
    }
    return(ok);
}
```

B.3 Crunch Subroutine

```
/*
```

```
 * crunch
 * Pete Ford, Durham University, June 1993
 * Picks apart spacegroup symbols to get the class, lattice and operation
 * symbols. Expects any subscripts to be in brackets or preceded by '_', but
 * otherwise accepts standard spacegroup symbols and is tolerant of spaces
 *
 * Common Blocks:
 */
#include <crack.h>
/*
 * Local defines:
 */
#define One    0
#define Two    1
#define Three  2
#define Four   3
#define Plane  4
#define Six    5
/*
 * Functions called:
 */
void point(char *);
void spgpex(char *, char *);
/*
 * Code
 */
LOGICAL crunch()
{
    int symlen, i, j, k, skip;
    char *pp, m[1024], bar3[Sgrpmax], *sp, stmp[Sgrpmax];
    LOGICAL flag[6][3], nottri, ok;
/*
 * Initialise variables to default values
 */

    ok = True;
    rgroup = True;
    *(parts.f) = NULL;
    for (i=0; i<3; i++)
    {
        parts.r[i][0] = NULL;
        parts.r[i][1] = NULL;
        for (j=0; j<6; j++)
        {
            flag[j][i] = False;
```

```
        }
    }
    nottri = False;
    pp = parts.f;
    if ((symlen = strlen(spgp.s)) == 0)
    {
        printf("crunch: No spacegroup symbol given\n");
        return (False);
    }
/*
 * Run the symbol given through spgpex to make interpretation easier
 */

    if (!strchr(spgp.s,'_'))
    {
        strcpy (stmp, spgp.s);
        spgpex (stmp, spgp.s);
    }
    symlen = strlen(spgp.s);

/*
 * Read each character in turn; i is character counter, j is axis counter
 */
    for (i=0, j=0; i<symlen && j<3 && ok; i++)
    {
        switch (spgp.s[i])
        {

/*
 * store centring symbol
 */
        case 'P': case 'A': case 'B': case 'C':
        case 'F': case 'I': case 'R': case 'H':
            centring = spgp.s[i];
            break;

/*
 * if a '1' then store that and move on to the next axis
 */

        case '1':
            parts.r[j][0] = pp;
            *pp++ = '1';
            *pp++ = NULL;
            flag[One][j] = True;
```

```
            j++;
            break;

/*
 * If a bar, then check that the next character is valid
 */

        case '-':
            i++;
            switch (spgp.s[i])
            {
            case '3': case '4': case '6':
                nottri = True;

            case '1':        /* Note intentional fall-through */
                parts.r[j][0] = pp;
                *pp++ = '-';
                *pp++ = spgp.s[i];
                *pp++ = NULL;
                k = spgp.s[i] - '1';
                flag[k][j] = True;
                j++;
                break;

            default:
                printf("crunch: Invalid rotation inversion spgp");
                return (False);
                break;
            }
            break;

/*
 * If a '2' then look for '(' or '_' for a subscript
 */

        case '2':
            parts.r[j][0] = pp;
            *pp++ = '2';
            switch (spgp.s[i+1])
            {
            case '_':
                if (spgp.s[i+2] == '1')
                {
                    *pp++ = '1';
                    i+=2;
```

```
            }
            else
            {
                printf("crunch: Invalid subscript for two-fold axis");
                return (False);
            }
            break;

        case '(':
            if (spgp.s[i+2] == '1' &&
            spgp.s[i+3] == ')')
            {
                *pp++ = '1';
                i+=3;
            }
            else
            {
                printf("crunch: Invalid subscript for two-fold axis");
                return (False);
            }
            break;

        default:
            break;
        }
        nottri = True;
        *pp++ = NULL;
        flag[Two][j] = True;
        j++;
        break;

/*
 * If a 3 is found then check the next character
 */

    case '3':
        parts.r[j][0] = pp;
        *pp++ = '3';
        switch (spgp.s[i+1])
        {
        case '_':
            if (spgp.s[i+2] == '1' ||
            spgp.s[i+2] == '2')
            {
                *pp++ = spgp.s[i+2];
```

```
            i+=2;
        }
        else
        {
            printf("crunch: Invalid subscript for three-fold axis");
            return (False);
        }
        break;

    case '(':
        if ((spgp.s[i+2] == '1' ||
        spgp.s[i+2] == '2') &&
        spgp.s[i+3] == ')')
        {
            *pp++ = spgp.s[i+2];
            i+=3;
        }
        else
        {
            printf("crunch: Invalid subscript for three-fold axis");
            return (False);
        }
        break;

    default:
        break;
    }
    nottri = True;
    *pp++ = NULL;
    k = spgp.s[i] - '0';
    flag[Three][j] = True;
    j++;
    break;

/*
 * If a 4 is found then check the possibilities
 */

case '4':
    parts.r[j][0] = pp;
    *pp++ = '4';
    switch (spgp.s[i+1])
    {
    case '_':
        if (spgp.s[i+2] == '1' ||
```

```
            spgp.s[i+2] == '2' ||
            spgp.s[i+2] == '3')
            {
                *pp++ = spgp.s[i+2];
                i+=2;
            }
            else
            {
                printf("crunch: Invalid subscript for four-fold axis");
                return (False);
            }
            break;

        case '(':
            if ((spgp.s[i+2] == '1' ||
            spgp.s[i+2] == '2' ||
            spgp.s[i+2] == '3') &&
            spgp.s[i+3] == ')')
            {
                *pp++ = spgp.s[i+2];
                i+=3;
            }
            else
            {
                printf("crunch: Invalid subscript for four-fold axis");
                return (False);
            }
            break;

        default:
            break;
        }
        nottri = True;
        *pp++ = NULL;
        k = spgp.s[i] - '0';
        flag[Four][j] = True;
        j++;
        break;

/*
 * If a 6 is found then check the possibilities
 */

    case '6':
        parts.r[j][0] = pp;
```

```
*pp++ = '6';
switch (spgp.s[i+1])
{
case '_':
    if (spgp.s[i+2] == '1' ||
    spgp.s[i+2] == '2' ||
    spgp.s[i+2] == '3' ||
    spgp.s[i+2] == '4' ||
    spgp.s[i+2] == '5')
    {
        *pp++ = spgp.s[i+2];
        i+=2;
    }
    else
    {
        printf("crunch: Invalid subscript for six-fold axis");
        return (False);
    }
    break;

case '(':
    if ((spgp.s[i+2] == '1' ||
    spgp.s[i+2] == '2' ||
    spgp.s[i+2] == '3' ||
    spgp.s[i+2] == '4' ||
    spgp.s[i+2] == '5') &&
    spgp.s[i+3] == ')')
    {
        *pp++ = spgp.s[i+2];
        i+=3;
    }
    else
    {
        printf("crunch: Invalid subscript for six-fold axis");
        return (False);
    }
    break;

default:
    break;
}
nottri = True;
*pp++ = NULL;
k = spgp.s[i] - '0';
flag[Six][j] = True;
```

```
          j++;
          break;

/*
 * If a / is found, check the flags for validity, and
 * backspace the axis counter to apply the plane processing to the
 * appropriate axis
 */

      case '/':
          if (flag[Two][j-1] ||
          flag[Four][j-1] ||
          flag[Six][j-1])
          {
              j--;
          }
          else
          {
              printf("crunch: Invalid axis order for perpendicular plane");
              return (False);
          }
          break;
/*
 * Consider possible glide plane symbols
 */

      case 'm': case 'a': case 'b': case 'c': case 'n': case 'd':
          parts.r[j][1] = pp;
          *pp++ = spgp.s[i];
          *pp++ = NULL;
          flag[Plane][j] = True;
          rgroup = False;
          nottri = True;
          j++;
          break;
/*
 * Special cases for rhombohedral groups
 */

      case 'r': case 'h':
          if (centring == 'R' || centring == 'H') break;
/*
 * ignore spaces but generate an error at anything else
 */
```

```
        default:
            if (spgp.s[i] && !isspace(spgp.s[i]))
            {
                printf("crunch: Invalid character in spacegroup %s",spgp.s);
                return (False);
            }
            break;
        }
    }
/*
 * From what has been extracted, it should be possible to determine the crystal
 * class of the space group using the flags set during the processing.
 *
 * If the special 'nottri' flag is clear then must be Triclinic (P1 or P-1)
 */
    if (!nottri)
    {
        strcpy(class.s, "Triclinic");
        return (True);
    }
/*
 * If the centring symbol is R then must be Rhombohedral
 */
    if (centring == 'R')
    {
        strcpy(class.s, "Rhombohedral");
        return (True);
    }
    if (centring == 'H')
    {
        strcpy(class.s, "Hexagonal");
        return (True);
    }

/*
 * If a six-fold was found then must be hexagonal
 */

    if (flag[Six][0] || flag[Six][1] || flag[Six][2])
    {
        strcpy(class.s, "Hexagonal");
        return (True);
    }
/*
 * If four-fold was found then check for a three-fold to decide between Cubic
```

```
 * and Tetragonal
 */
   if (flag[Four][0])
   {
       if (flag[Three][1])
       {
           strcpy(class.s, "Cubic");
       }
       else
       {
           strcpy(class.s, "Tetragonal");
       }
       return (True);
   }

/*
 * If a three-fold is found in the second axis and the others are not ones then
 * it must be Cubic
 */
   if (flag[Three][1] && !flag[One][0])
   {
       strcpy(class.s, "Cubic");
/*
 * Correct for old-style symbol in Cubic groups where '3' should be '-3'
 * by checking with the point group
 */

       point(bar3);
       if (!strcmp(bar3,"m3") || !strcmp(bar3,"m3m"))
       {
           sp = stmp+1;
           strcpy(sp,spgp.s);
           for (i=0; (stmp[i] = spgp.s[i]) != '3'; i++);
           stmp[i] = '-';
           strcpy(spgp.s,stmp);
           printf("Spacegroup symbol corrected to %s\n",spgp.s);
           return (crunch());
       }
       return (True);
   }
/*
 * If a three-fold was found (other than Cubic or Hexagonal) then must be
 * Trigonal
 */
   if (flag[Three][0] || flag[Three][1] || flag[Three][2])
```

```
    {
        strcpy(class.s, "Trigonal");
        return (True);
    }
/*
 * If all the axes are either two-folds or planes then it's Orthorhombic
 */
    if ((flag[Two][0] || flag[Plane][0]) &&
    (flag[Two][1] || flag[Plane][1]) &&
    (flag[Two][2] || flag[Plane][2]))
    {
        strcpy(class.s, "Orthorhombic");
        return (True);
    }
/*
 * Otherwise, must be Monoclinic; this is the hardest to test, so find by
 * elimination!
 */

    strcpy(class.s, "Monoclinic");
    return (True);
}
```

B.4 Detntr Subroutine

```
/*
 * detntr
 *
 * Pete Ford, Durham University, June 1992
 * Calculates the determinant and trace of a 3x3 integer matrix. Useful for
 * testing the type of rotation matrix (see Giaccovazzo, Oxford 1991, p42)
 *
 *
 * Common Blocks:
 */
/*
 * Local variables:
 */
#define I2D(ptr,x,y) *(ptr+3*x+y)
/*
 * Functions called:
 */
```

```
/*
 *
 */
void detntr(int *imat, int *det, int *trace)
{
    *trace = I2D(imat,0,0) + I2D(imat,1,1) + I2D(imat,2,2);
    *det = (I2D(imat,0,0) * I2D(imat,1,1) * I2D(imat,2,2) +
         I2D(imat,1,0) * I2D(imat,0,2) * I2D(imat,2,1) +
         I2D(imat,2,0) * I2D(imat,0,1) * I2D(imat,1,2))
         - (I2D(imat,0,0) * I2D(imat,1,2) * I2D(imat,2,1) +
         I2D(imat,1,0) * I2D(imat,0,1) * I2D(imat,2,2) +
         I2D(imat,2,0) * I2D(imat,0,2) * I2D(imat,1,1));
    return;
}
```

B.5 Invpt Function

```
/*
 * invpt
 * Pete Ford, Durham University, June 1993
 * Finds an inversion (if one exists) in the spacegroup symbol matrices
 * Only the rotational parts of these matrices are needed, since the
 * translations will be consistent eventually in any case.
 *
 *
 * Common Blocks:
 */
#include <crack.h>
/*
 * Local variables:
 */
/*
 * Functions called:
 */
void detntr(int *, int *, int *);
/*
 *
 */
LOGICAL invpt(void)
{
    int rmats[11][3][3], tmat[3][3], wmat[3][3], *tp, ptr[11];
    int i, j, k, n, trace[11], det[11], ttr, tdet, temp, first, second;
/*
```

```
 * Convert spacegroup matrices into rotation matrices.
 */
   tp = *tmat;
   for (k=0; k<11; k++)
   {
      for (j=0; j<3; j++)
      {
         for (i=0; i<3; i++)
         {
            tmat[i][j] = rmats[k][i][j] = (int) matrix[k].m[i][j];
         }
      }
      detntr(tp, &det[k], &trace[k]);
/*
 * The special case is a -3 axis, where the method below doesn't work; ALL
 * spacegroups with -3 are centrosymmetric.
 *
 * Test for a -3 symbol by it's determinant and trace
 */
      if (!trace[k] && det[k] == -1) return (True);
/*
 * Also test for a -1 (spacegroup P-1) to save time
 */

      if (trace[k] == -3) return (True);
   }
/*
 * set up an order matrix based on the determinant and trace of the operations
 */

   for (i=0; i<11; ptr[i]=i, i++);
   for (i=0; i<10; i++)
   {
      for (j=i; j<11; j++)
      {
         if (trace[i] * det[i] > trace[j] * det[j])
         {
            temp = ptr[i];
            ptr[i] = ptr[j];
            ptr[j] = temp;
         }
      }
   }
/*
 * Multiply the rotation matrices together in order until an
```

```
 * inversion centre appears
 */

    for (k=0; trace[k]==3; k++);
    for (i=0; i<3; i++) for (j=0; j<3; j++) wmat[i][j] = rmats[k][i][j];
    for (n=k+1; n<11; n++)
    {
        if (trace[n] != 3)
        {
            for (i=0; i<3; i++)
            {
                for (j=0; j<3; j++)
                {
                    tmat[i][j] = wmat[i][0] * rmats[n][0][j] +
                                 wmat[i][1] * rmats[n][1][j] +
                                 wmat[i][2] * rmats[n][2][j];
                }
            }
            detntr(tp, &tdet, &ttr);
            if (ttr == -3)
            {
                return (True);
            }
            for (i=0; i<3; i++) for (j=0; j<3; j++) wmat[i][j] = tmat[i][j];
        }
    }

/*
 * If an inversion existed it should have been found, otherwise there isn't one
 */

    return (False);
}
```

B.6 Matax Subroutine

```
/*
 * matax
 * Pete Ford, Durham University, June 1993
 * Uses the parts array and the class to determine the matrices for the
 * operators specified by the spacegroup symbol. Includes an identity as the
 * first matrix each time.
 *
```

```
 *
 * Common Blocks:
 */
#include <crack.h>
#include <matrix.h>
/*
 * Local variables:
 */
static char *bit[] = {"st","nd","rd"};
static int order[] = {0,0,0,0,0,0,0,0,0,0,1,2,1,1,2,2,2,2,2,4,4,4,4,4,4,4};
/*
 * Functions called:
 */
int symget(int, int, int, int);
/*
 *
 */
int matax(LOGICAL multi)
{
    char m[1024];
    MATRIX wkmat;
    LOGICAL ok;
    int i, j, k, mret, n, x ,y, cno, iops[3][2];
    for (k=0; k<11; k++)
    {
        for (j=0; j<4; j++)
        {
            for (i=0; i<4; i++)
            {
                matrix[k].m[i][j] = (i==j) ? 1 : 0;
            }
            matrix[k].f[i] = NULL;
        }
    }
    ok = True;

/*
 * Find the class identifier among the eight possible classes
 */

    for (cno=0; *cident[cno] && strncmp(cident[cno],class, 4); cno++);
    if (! *cident[cno])
    {
        printf("matax: fatal error");
        return (0);
```

```
    }

/*
 * Go through the parts array to find which symbols are present
 */

    for (i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            if (parts.r[i][j])
            {
                for (k=0; *ops[k] && strcmp(ops[k],parts.r[i][j]); k++);
                iops[i][j] = (*ops[k]) ? k : 0;
            }
            else
            {
                iops[i][j] = 0;
            }
        }
    }
/*
 * If class is Monoclinic, and the symbol is ambiguous, then use the b-axis if
 * possible, of the c-axis is there is a b-glide
 */

    if (cno == 1 && !iops[1][0] && !iops[1][1])
    {
        if (iops[0][1] != B_glide)
        {
            iops[1][0] = iops[0][0]; parts.r[1][0] = parts.r[0][0];
            iops[1][1] = iops[0][1]; parts.r[1][1] = parts.r[0][1];
            iops[0][0] = 0; parts.r[0][0] = NULL;
            iops[0][1] = 0; parts.r[0][1] = NULL;
        }
        else
        {
            iops[2][0] = iops[0][0]; parts.r[2][0] = parts.r[0][0];
            iops[2][1] = iops[0][1]; parts.r[2][1] = parts.r[0][1];
            iops[0][0] = 0; parts.r[0][0] = NULL;
            iops[0][1] = 0; parts.r[0][1] = NULL;
        }
    }
/*
 * Look at each part of the symbol in turn, and use the class to decide which
```

```
 * axis or vector the symbol applies to. At the same time do a check to see if
 * the operation is valid on that axis.
 */

    n = 1;
    for (i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            if (iops[i][j])
            {
                for (x=0; x<4; x++)
                {
                    for (y=0; y<4; y++)
                    {
                        mret = symget(iops[i][j],axisno[cno][i],x,y);
                        if (mret == 12)
                        {
                            mret = 0;
                            matrix[n].f[x] = 1;
                        }
                        matrix[n].m[x][y] = mret;
                    }
                }
                if (matrix[n].m[3][3] == 0)
                {
                    sprintf(m,"Operation %s not valid for the %d%s symbol in %s spacegroups",
                    ops[iops[i][j]],i+1,bit[i],class);
                    printf(m);
                    return (0);
                }
/*
 * If multi is set TRUE then the full set of operations for high order axes is
 * required. If any of the symbols are of order greater than 2, then duplicate
 * them enough times to reveal any hidden symmetry operators.
 */
                if (multi)
                {
                    if (order[iops[i][j]])
                    {
                        for (x=0; x<4; x++)
                        {
                            for (y=0; y<0; y++)
                            {
```

```
                        wkmat.m[x][y] = matrix[n].m[x][y];
                    }
                    wkmat.f[x] = matrix[n].f[x];

                }
                for (k=0; k<order[iops[i][j]]; k++)
                {
                    for (x=0; x<4; x++)
                    {
                        for (y=0; y<0; y++)
                        {
                            matrix[n+1].m[x][y] = matrix[n].m[x][0] * wkmat.m[0][y]
                                                + matrix[n].m[x][1] * wkmat.m[1][y]
                                                + matrix[n].m[x][2] * wkmat.m[2][y]
                                                + matrix[n].m[x][3] * wkmat.m[3][y];
                        }
                        matrix[n+1].f[x] = matrix[n].f[x] | wkmat.f[x];
                    }
                    n++;
                }                       /* for k....    */
            }                           /* if order... */
        }                                   /* if multi     */
        n++;
    }                                       /* if iops.... */
  }                                             /* for j....    */
}                                               /* for i....    */
return (n);
}
```

B.7 Onemat Subroutine

```
/*
 * onemat
 *
 * Pete Ford, Durham Unversity, June 1993
 * Gets the matrix for one symbol in the given class
 * Modified form of MATAX routine
 *
 *
 * Common Blocks:
 */
#include <crack.h>
#include <matrix.h>
```

```
/*
 * Local variables:
 */
#define axes 10
static MATRIX zero =
                   {
{NULL, NULL, NULL, NULL},
    {
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0},
    {0,0,0,0}
    }
};
/*
 * Functions called:
 */
int symget(int, int, int, int);

/*
 *
 */
MATRIX onemat(char *axsym, int ax)
{
    char m[1024];
    MATRIX matr;
    int i, j, k, n, cno;
/*
 * Initialise
 */

    for (j=0; j<4; j++)
    {
        for (i=0; i<4; i++)
        {
            matr.m[i][j] = (i == j) ? 1 : 0;
        }
    }

/*
 * if the axis symbol is NULL, then return an identity matrix
 */

    if (! *axsym) return (matr);
/*
```

```
 * Determine the class number to identify which matrix to use, in conjunction
 * with the axis number ax
 */

    for (cno=0; *cident[cno] && strncmp(cident[cno],class, 4); cno++);
    if (! *cident[cno])
    {
        printf("onemat: Spacegroup class not known");
        return (zero);
    }
/*
 * Find out what the symbol is
 */
    for (k=0; *ops[k] && strcmp(ops[k],axsym); k++);
    if (! *ops[k])
    {
        printf("onemat: axis symbol not known");
        return (zero);
    }
/*
 * get the matrix for the symbol on the axis given by ax and cno
 */

    if (symget(k,axisno[cno][ax],3,3) == 0)
    {
        printf("onemat: symbol %s in position %d is not valid",axsym, ax);
        return (zero);
    }

    for (i=0; i<4; i++)
    {
        for (j=0; j<4; j++)
        {
            matr.f[i] = NULL;
            n = symget(k,axisno[cno][ax],i,j);
            if (n == 12)
            {
                n = 0;
                matr.f[i] = 1;
            }
            matr.m[i][j] = n;
        }
    }

    return (matr);
```

```
}
```

B.8 Point Subroutine

```
/*
 * point
 *
 * Pete Ford, Durham University, June 1993
 * Makes a point group out of the spacegroup symbol, by simple one-to-one mapping
 *
 *
 * Common Blocks:
 */
#include <crack.h>
/*
 * Local variables:
 */
static char *sops[] =
                {
    "1","-1","2","m","21","a","b","c","n",
    "d","3","-3","31","32","4","-4","41","42",
    "43","6","-6","61","62","63","64","65",NULL
};
static char *pops[] =
                {
    "1","-1","2","m","2","m","m","m","m",
    "m","3","-3","3","3","4","-4","4","4",
    "4","6","-6","6","6","6","6","6",NULL
};
/*
 * Functions called:
 */

/*
 *
 */
void point(char *ptgrp)
{
    int i, j, n;
    LOGICAL rotn;
    char *pp;
/*
 * Look at each of the parts in turn and convert spacegroup symbols into point
```

```
 * group operations
 */

    pp = ptgrp;
    *pp = NULL;
    for (i=0; i<3; i++)
    {
        rotn = False;
        for (j=0; *sops[j] && strcmp(sops[j],parts.r[i][0]); j++);
        if (j<26)
        {
            rotn = True;
            *pp++ = pops[j][0];
            if (pops[j][1]) *pp++ = pops[j][1];
        }
        for (j=0; *sops[j] && strcmp(sops[j],parts.r[i][1]); j++);
        if (j<26)
        {
            if (rotn)
            {
                *pp++ = '/';
            }
            *pp++ = pops[j][0];
            if (pops[j][1]) *pp++ = pops[j][1];
        }
        *pp = NULL;
    }
    return;
}
```

B.9 Pretty Subroutine

```
/*
 * pretty
 *
 * Pete Ford, Durham University, June 1993
 * Converts a matrix of numbers into a string representing the equivalent
 * position in x,y,z form
 *
 *
 * Common Blocks:
 */
#include <crack.h>
```

```
#include <math.h>
/*
 * Local variables:
 */
static char *trans[] =
                 {
   NULL,"1/12","1/6","1/4","1/3","5/12",
   "1/2","7/12","2/3","3/4","5/6","11/12"
};

static char pts[3][10];

static char *axis="xyz";
#define I2D(m,x,y) *(m+x*4+y)
/*
 * Functions called:
 */

/*
 *
 */
void pretty(int *pmat, char *output)
{
   int intmat[4][4], i, j, k, n;
   int tmp;
   char sign, temp[80], *pp, *tp;
/*
 * Convert the input matrix into an array for easier handling
 */

   for (i=0; i<3; i++)
   {
      for (j=0; j<3; j++)
      {
         intmat[i][j] = I2D(pmat,i,j);
      }
      intmat[i][3] =  (I2D(pmat,i,j) + 144) % 12;
/*
 * Now work out the rotational pts of the matrix in x,y,z form for each axis
 */

      pp = pts[i];
      n = 0;
      for (j=0; j<3; j++)
      {
```

```
            switch (intmat[i][j])
            {
            case -1:
                *pp++ = '-';
                *pp++ = axis[j];
                break;
            case 1:
                *pp++ = '+';
                *pp++ = axis[j];
                break;
            default:
                break;
            }
        }
/*
 * Add on the translational part of the matrix
 */

        tp = trans[intmat[i][3]];
        if (*tp)
        {
            *pp++='+';
            for (k=0; tp[k]; k++) *pp++ = tp[k];

        }
        *pp = NULL;
    }
    sprintf(output,"%s,%s,%s",pts[0],pts[1],pts[2]);
    return;
}
```

B.10 Selgen Subroutine

```
/*
 * selgen
 *
 * Version 3.1; Pete Ford, Durham University, October 1995
 * Selects and adjusts the generators that will create the spacegroup. Apologies
 * for the poor commentary, but it's largely based on empirical rules derived
 * from a study of International Tables Volume A.
 *
 *
 * Common Blocks:
```

```
 */
#include <crack.h>
#include <math.h>
/*
 * Local variables:
 */
static char *pref[] =
                    {
   "-1","43","42","41", "4","-4","61","62","63",
   "64","65", "6","-6", "n", "c", "b", "a", "d",
   "21", "m", "2","31","32", "3","-3",NULL
};
#define N_PREF 25
static char *classes[] =
                       {
   "Triclinic",
   "Monoclinic",
   "Orthorhombic",
   "Tetragonal",
   "Rhombohedral",
   "Trigonal",
   "Hexagonal",
   "Cubic",
   NULL
};
#define cchs "ABCIH"
#define spch "abc"
/*
 * Functions called:
 */
void point(char *);
void toppri(int *, int *, int *, int *, int *);
MATRIX onemat(char *, int);
int detntr(int *, int *, int *);
/*
 *
 */
int selgen(void)
{
   char ptgrp[6], spec[6];
   int i, j, k, n, k1, m1, n1, k2, m2, n2, x, y,
   pno[4][2], ngen, cl, imat[3][3], det, trace, test, *pptr, *iptr;
   LOGICAL genflg[4][2];
   MATRIX wg[4][2], wkmat[5], tmat;
   double shift[3], tshift;
```

```
/*
 * gmats, wkmat and wg need initialising to identity matrices
 */

    for (i=0; i<4; i++)
    {
        for (j=0; j<4; j++)
        {
            tshift = (i == j) ? 1 : 0;
            for (k=0; k<11; k++)
            {
                if (k < 5)
                {
                    wkmat[k].m[i][j] = tshift;
                }
                if (k < 4)
                {
                    wg[k][0].m[i][j] = tshift;
                    wg[k][1].m[i][j] = tshift;
                }
                gmats[k].m[i][j] = tshift;
            }
        }
    }
    for (i=0; i<3; i++)
    {
        for (k=0; k<11; k++)
        {
            if (k < 5)
            {
                wkmat[k].f[i] = NULL;
            }
            if (k < 4)
            {
                wg[k][0].f[i] = NULL;
                wg[k][1].f[i] = NULL;
            }
            gmats[k].f[i] = NULL;
        }
    }

    shift[0] = shift[1] = shift[2] = 0;
/*
 * For some types, it will be helpful to know the point group
 */
```

```
    point(ptgrp);
/*
 * Go through all of the parts of the spacegroup symbol
 */

    for (i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            genflg[i][j] = False;
/*
 * Get the preference number for each symbol present;
 */

            for (k=0; *pref[k] && strcmp(parts.r[i][j],pref[k]); k++);
            if ((pno[i][j] = k) == N_PREF) genflg[i][j] = False;
/*
 * Store all of the matrices in wg
 */
            tmat = onemat(parts.r[i][j],i);
            for (x=0; x<4; x++)
            {
                for (y=0; y<4; y++)
                {
                    wg[i][j].m[x][y] = tmat.m[x][y];
                }
                wg[i][j].f[x] = tmat.f[x];
            }
        }
    }
    pno[3][0] = N_PREF;
    pno[3][1] = N_PREF;
    genflg[3][0] = False;
    genflg[3][1] = False;
    pptr = *pno;
    iptr = *imat;
/*
 * If the spacegroup is centrosymmetric then set wg[3][0] to the inversion
 */
    if (centric && strcmp(ptgrp,"-1"))
    {
        for (i=0; i<3; i++)
        {
            wg[3][0].m[i][i] = -1;
```

```
            wg[3][0].m[i][3] = 0;
        }
        pno[3][0] = 1;
        genflg[3][0] = True;
    }

    for (cl=0; *classes[cl] && strcmp(classes[cl],class.s); cl++);
    switch (cl)
    {
/*
 * For a Monoclinic cell...
 */

    case 1:
/*
 * The positions are given by making all of the translations the same if they
 * are flagged as variable
 */

        for (i=0; i<4; i++)
        {
            for (j=0; j<2; j++)
            {
                for (k=0; k<3; k++)
                {
                    tshift = abs(wg[i][j].m[k][3] % 12);
                    if (tshift > shift[k]) shift[k] = tshift;
                }
            }
        }
        for (i=0; i<4; i++)
        {
            for (j=0; j<2; j++)
            {
                for (k=0; k<3; k++)
                {
                    if (wg[i][j].f[k]) wg[i][j].m[k][3] = shift[k];
                }
            }
        }
/*
 * Mark the two highest priority operators as generators
 */
        pptr = *pno;
        toppri(pptr,&i,&j,&x,&y);
```

```
        if (pno[3][0] < N_PREF)      /* toppri doesn't look at pno[3][] */
        {                            /* so if inversion exists force it */
            x=i; y=j;                /* to be a generator */
            i=3; j=0;
        }
        genflg[i][j] = True;
        genflg[x][y] = True;
        break;
/*
 * For an Orthorhombic cell...
 */

    case 2:
/*
 * Deal with rotation-only systems - these are tricky
 */
        if (!strcmp(ptgrp,"222"))
        {
            genflg[0][0] = True;
            genflg[1][0] = True;
            test = pno[0][0] + pno[1][0] + pno[2][0];
        }
        if (test == 58)          /* Spacegroup 222(1) */
        {
            wg[0][0].m[1][3] = 0;
            wg[0][0].m[2][3] = 0;
            for (i=0; i<3; i++)
            {
                if (wg[1][0].f[i])
                {
                    wg[1][0].m[i][3] = wg[0][0].m[i][3] +
                                       wg[1][0].m[i][3] +
                                       wg[2][0].m[i][3];
                }
            }
        }
        else if (test == 56)     /* Spacegroup 2(1)2(1)2 */
        {
            for (i=0; i<3; i++)
            {
                if (!strcmp(parts.r[i][0],"21"))
                {
                    for (j=0; j<3; j++)
                    {
                        if (wg[i][0].f[j])
```

```
                        {
                            wg[i][0].m[j][3] = wg[0][0].m[j][3] +
                                               wg[1][0].m[j][3] +
                                               wg[2][0].m[j][3];
                        }
                    }
                }
            }
        }
        else if (test == 54)      /* Spacegroup 2(1)2(1)2(1) */
        {
            wg[0][0].m[1][3] = 6;
            wg[1][0].m[2][3] = 6;
        }
/*
 * Point group mm2
 */

        else if (!strcmp(ptgrp,"mm2") ||
        !strcmp(ptgrp,"m2m") ||
        !strcmp(ptgrp,"2mm"))
        {
            toppri(pptr,&m1,&m2,&n1,&n2);
            genflg[m1][m2] = True;
            genflg[n1][n2] = True;
            k1 = 3 - (m1 + n1);
            k2 = (pno[k1][1] < N_PREF) ? 0 : 1;
            for (i=0; i<3; i++)
            {
                if (wg[m1][m2].f[i])
                {
                    wg[m1][m2].m[i][3] = wg[n1][n2].m[i][3] + wg[k1][k2].m[i][3];
                }
                if (wg[n1][n2].f[i])
                {
                    wg[n1][n2].m[i][3] = wg[m1][m2].m[i][3] + wg[k1][k2].m[i][3];
                }
            }
        }
/*
 * Point group mmm
 */
        else if (!strcmp(ptgrp,"mmm"))
        {
            toppri(pptr,&m1,&m2,&n1,&n2);
```

```
        genflg[m1][m2] = True;
        genflg[n1][n2] = True;
        k1 = 3 - (m1 + n1);
        k2 = (pno[k1][0] < N_PREF) ? 0 : 1;
/*
 * Check for special cases
 */

        sprintf(spec,"%c%c%c",*parts.r[m1][m2],*parts.r[n1][n2],*parts.r[k1][k2]);
        if (!strcmp(spec,"amm") ||
        !strcmp(spec,"bmm") ||
        !strcmp(spec,"cmm"))
        {
            for(n=0; *cchs && centring != cchs[n]; n++);
            for(x=0; *spch && *(parts.r[m1][m2]) != spch[x]; x++);
            if (n == 3)
            {
                for (i=0; i<3; i++)
                {
                    if (wg[m1][m2].f[i])
                    {
                        wg[m1][m2].m[i][3] = wg[m1][m2].m[i][3] +
                                             cvecs[n][i];
                    }
                }
            }
            else if (n != 0 && x != n)
            {
                for (i=0; i<3; i++)
                {
                    wg[m1][m2].m[i][3] = wg[m1][m2].m[i][3] +
                                         cvecs[n][i];
                }
            }
        }
        else if ((!strcmp(spec,"cam")||!strcmp(spec,"bam")||!strcmp(spec,"cbm"))
        && (centring != 'P' && centring != 'I'))
        {
            for (n=0; *cchs && centring != cchs[n]; n++);
            for (i=0; i<3; i++)
            {
                wg[n1][n2].m[i][3] = wg[n1][n2].m[i][3] + cvecs[n][i];
                wg[m1][m2].m[i][3] = wg[m1][m2].m[i][3] + cvecs[n][i];
            }
        }
```

```
/*
 * Do next bit for all cases except Fddd
 */

          if (strcmp(spec,"ddd"))
          {
             for (i=0; i<3; i++)
             {
                if (wg[m1][m2].f[i])
                {
                   wg[m1][m2].m[i][3] = wg[n1][n2].m[i][3]
                                      + wg[m1][m2].m[i][3]
                                      + wg[k1][k2].m[i][3];
                }
                if (wg[n1][n2].f[i])
                {
                   wg[n1][n2].m[i][3] = wg[n1][n2].m[i][3]
                                      + wg[m1][m2].m[i][3]
                                      + wg[k1][k2].m[i][3];
                }
             }
          }
       }
       break;
/*
 * For a Tetragonal cell...
 */

    case 3:
/*
 * Point groups 4 and -4 are simple; except for I4(1) where the 4-fold is off
 * the origin
 */

       if (!strcmp(ptgrp,"4") || !strcmp(ptgrp,"-4"))
       {
          for (i=0; i<4; i++)
          {
             for (j=0; j<2; j++)
             {
                wg[i][j].f[0] = NULL;
                wg[i][j].f[1] = NULL;
                wg[i][j].f[2] = NULL;
                if (!strcmp(parts.r[0][0],"41") && centring == 'I')
                {
```

```
                wg[i][j].m[1][3] = 6;
            }
        }
    }
/*
 * Mark any operator that is not an identity as a generator
 */

    for (i=0; i<4; i++)
    {
        for (j=0; j<2; j++)
        {
            for (x=0; x<3; x++)
            {
                for (y=0; y<3; y++)
                {
                    imat[x][y] = wg[i][j].m[x][y];
                }
            }
            detntr(iptr, &det, &trace);
            genflg[i][j] = (trace != 3) ? True : False;
        }
    }
}
/*
 * Point group 4/m
 */

else if (!strcmp(ptgrp,"4/m"))     /* All this is empirical */
{
    wg[0][0].m[0][3] = wg[0][1].m[0][3] + wg[0][0].m[2][3];
    wg[0][0].m[1][3] = wg[0][0].m[2][3];
    wg[0][1].m[2][3] = 2 * wg[0][0].m[2][3];
    genflg[0][0] = genflg[0][1] = True;
}

/*
 * Point group 422 next...
 */
else if (!strcmp(ptgrp,"422"))
{
    if (!strcmp(parts.r[1][0],"21"))
    {
        wg[0][0].m[0][3] = 6;     /* This moves the 4-fold if the    */
        wg[0][0].m[1][3] = 6;     /* second symbol is a 2(1)         */
```

```
            wg[1][0].m[1][3] = 6;      /* Also the 2(1) is moved        */
            wg[1][0].m[2][3] = 12 - wg[0][0].m[2][3];
        }
        else
        {
            if (!strcmp(parts.r[0][0],"4") || !strcmp(parts.r[0][0],"42"))
            {
                wg[1][0].m[2][3] = 0;    /* The 2-fold is not shifted */
            }                            /* for 4 and 4(2) */
            else
            {
                wg[1][0].m[2][3] = 6;    /* but shifted 1/2 in z for 4(1),4(3) */
            }
        }
        if (centring == 'I' && !strcmp(parts.r[0][0],"41"))
        {
            wg[0][0].m[1][3] = 6;      /* I4(1)22 is a special case */
            wg[1][0].m[1][3] = 6;        /* and needs some adjustment */
            wg[1][0].m[2][3] = 3;
        }
        genflg[0][0] = genflg[1][0] = True;
    }
/*
 * Point group 4mm
 */

    else if (!strcmp(ptgrp,"4mm"))
    {

/*
 * In most cases the 4-fold goes on the origin, but some special cases
 * exist
 */
        if (!strcmp(parts.r[0][0],"42") && !strcmp(parts.r[1][1],"n"))
        {
            wg[0][0].m[0][3] = 6;
            wg[0][0].m[1][3] = 6;
        }
        else if (centring == 'I' && !strcmp(parts.r[0][0],"41"))
        {
            wg[0][0].m[1][3] = 6;
        }
        wg[1][1].m[0][3] = wg[1][1].m[1][3]; /* Empirical rule for mirror plane */
        genflg[0][0] = genflg[1][1] = True;
    }
```

```
/*
 * Point group -42m/-4m2
 */

        else if (!strcmp(ptgrp,"-42m") || !strcmp(ptgrp,"-4m2"))
        {
            toppri(pptr,&n1,&n2,&m1,&m2);    /* get the position of the m */
            k1 = 3 - (m1 + n1);              /* and that of the 2 */
            k2 = 0;
            if (m1 == 1)
            {
                wg[m1][m2].m[0][3] = wg[m1][m2].m[1][3];
            }
            else
            {
                wg[k1][k2].m[1][3] = wg[k1][k2].m[0][3];
            }
            wg[m1][m2].m[0][3] = wg[m1][m2].m[0][3] + wg[k1][k2].m[0][3];
            wg[m1][m2].m[1][3] = wg[m1][m2].m[1][3] + wg[k1][k2].m[1][3];
            if (!strcmp(parts.r[m1][m2],"d"))
            {
                wg[m1][m2].m[0][3] = 6;
                wg[m1][m2].m[1][3] = 0;
                wg[m1][m2].m[2][3] = 9;
            }
            genflg[0][0] = genflg[m1][m2] = True;
        }
/*
 * Point group 4/mmm
 */
        else if (!strcmp(ptgrp,"4/mmm"))
        {
            for (i=0; i<3; spec[i] = *(parts.r[i][1]), i++); spec[i] = NULL;
            if (!strcmp(spec,"mnm") && !strcmp(parts.r[0][0],"42"))
            {
                wg[0][0].m[0][3] = 6;    /* P4(2)/mnm is a special case for */
                wg[0][0].m[1][3] = 6;       /* some reason. */
            }
            else if (!strcmp(parts.r[0][1],"n"))
            {
                wg[0][0].m[0][3] = 6;       /* If the plane ppdr to c is an */
                wg[0][0].m[1][3] = 0;       /* n glide the 4-fold is on 1/4,1/4,z */
            }
            else if (!strcmp(parts.r[0][0],"41") && centring == 'I')
            {
```

```
            wg[0][0].m[0][3] = 3;     /* I4(1)/mmm groups have shifted 4-fold */
            wg[0][0].m[1][3] = 9;
        }
        else
        {
            wg[0][0].m[0][3] = 0;        /* All others have 4-fold on the origin */
            wg[0][0].m[1][3] = 0;
        }
        if (!strcmp(parts.r[0][0],"41") && centring == 'I')
        {
            wg[0][1].m[2][3] = 6;        /* I4(1)/mmm has first plane on 1/4 in z */
        }
        else
        {
            wg[0][1].m[2][3] = 0;
        }
        spec[2] = NULL;
        if (!strcmp(spec,"mb") || !strcmp(spec,"mn") ||
        !strcmp(spec,"nm") || !strcmp(spec,"nc"))
        {
            wg[1][1].m[0][3] = 6;
        }
        else
        {
            wg[1][1].m[0][3] = 0;
        }
        genflg[0][0] = genflg[0][1] = genflg[1][1] = True;
    }
    break;
/*
 * For a Trigonal cell...
 */

  case 5:

/*
 * The three-fold is always through the origin on the c-axis, so set the
 * a & b translations to zero
 */
    wg[0][0].m[0][3] = 0;
    wg[0][0].m[1][3] = 0;
/*
 * If it is -3 put the inversion point on the origin in the c direction
 */
```

```
        if (centric) wg[0][0].m[2][3] = 0;
/*
 * If the 3-fold is a screw axis, any two-folds need their z-location
 * adjusting. This should be possible in a general way since only the
 * two-folds will have their z-axis flagged for variable translation
 */

        if (wg[0][0].m[2][3] == 4)
        {                                          /* 3(1) axis */
            if (wg[1][0].f[2]) wg[1][0].m[2][3] = 8;
            if (wg[2][0].f[2]) wg[2][0].m[2][3] = 8;
        }
        else if (wg[0][0].m[2][3] == 8)
        {                                          /* 3(2) axis */
            if (wg[1][0].f[2]) wg[1][0].m[2][3] = 4;
            if (wg[2][0].f[2]) wg[2][0].m[2][3] = 4;
        }
/*
 else
 {
 if (wg[1][0].f[2]) wg[1][0].m[2][3] = 0;
 if (wg[2][0].f[2]) wg[2][0].m[2][3] = 0;
 }
 */
/*
 * Mark any operator that is not an identity as a generator
 */

        for (i=0; i<4; i++)
        {
            for (j=0; j<2; j++)
            {
                for (x=0; x<3; x++)
                {
                    for (y=0; y<3; y++)
                    {
                        imat[x][y] = wg[i][j].m[x][y];
                    }
                }
                detntr(iptr, &det, &trace);
                genflg[i][j] = (trace != 3) ? True : False;
            }
        }
        break;
/*
```

```
 * For a hexagonal cell...
 */
    case 6:

/*
 * 6/m, 6/mmm and 622 point groups have complications:
 */
        if (!strcmp(ptgrp,"6/m") ||
        !strcmp(ptgrp,"622") ||
        !strcmp(ptgrp,"6/mmm"))
        {
            tshift = 12 - abs(wg[0][0].m[2][3]);
            if (wg[0][1].f[2]) wg[0][1].m[2][3] = tshift;
            if (wg[2][0].f[2]) wg[2][0].m[2][3] = tshift;
        }
/*
 * Also in -6m2 and -62m the inversions don't necessarily lie on the origin
 */
        else if ((!strcmp(ptgrp,"-6m2") || !strcmp(ptgrp,"-62m")) &&
        (!strcmp(parts.r[1][1],"c") || !strcmp(parts.r[2][1],"c")))
        {
            wg[0][0].m[2][3] = 6;
        }
/*
 * Mark any operator that is not an identity as a generator
 */

        for (i=0; i<4; i++)
        {
            for (j=0; j<2; j++)
            {
                for (x=0; x<3; x++)
                {
                    for (y=0; y<3; y++)
                    {
                        imat[x][y] = wg[i][j].m[x][y];
                    }
                }
                detntr(iptr, &det, &trace);
                genflg[i][j] = (trace != 3) ? True : False;
            }
        }
        break;
/*
 * For a Cubic cell...
```

```
 */

   case 7:

/*
 * Point groups 23 and m-3
 */

      if (!strcmp(ptgrp,"23"))
      {
         wg[0][0].m[0][3] = wg[0][0].m[2][3];
         genflg[0][0] = genflg[1][0] = True;
      }
      else if (!strcmp(ptgrp,"m-3"))
      {
         wg[0][1].m[2][3] = wg[0][1].m[0][3] - wg[0][1].m[1][3];
         genflg[0][1] = genflg[1][0] = True;
      }
/*
 * Point group 432
 */

      else if (!strcmp(ptgrp,"432"))
      {
         if (centring == 'F' && !strcmp(parts.r[0][0],"41"))
         {                                /* Special case for F4(1)32 */
            wg[0][0].m[0][3] = wg[0][0].m[1][3] = 9;
            wg[2][0].m[0][3] = wg[2][0].m[1][3] = wg[2][0].m[2][3] = 3;
         }
         else
         {
            wg[0][0].m[0][3] = wg[0][0].m[2][3];
            wg[2][0].m[0][3] = wg[0][0].m[1][3] = 12 - wg[0][0].m[2][3];
            wg[2][0].m[2][3] = wg[2][0].m[1][3] = 12 - wg[0][0].m[2][3];
         }
         genflg[0][0] = genflg[1][0] = genflg[2][0] = True;
      }
/*
 * Point group -43m
 */

      else if (!strcmp(ptgrp,"-43m"))
      {
         if (centring == 'I' && !strcmp(parts.r[2][1],"d"))
         {
```

```
            wg[0][0].m[0][3] = wg[0][0].m[2][3] = 9;
            wg[0][0].m[1][3] = wg[2][1].m[0][3] = 3;
            wg[2][1].m[1][3] = wg[2][1].m[2][3] = 3;
        }
        else
        {
            wg[0][0].m[0][3] = wg[0][0].m[1][3] = wg[2][1].m[2][3];
            wg[0][0].m[2][3] = wg[2][1].m[0][3] = wg[2][1].m[2][3];
            wg[2][1].m[1][3] = wg[2][1].m[2][3];
        }
        genflg[0][0] = genflg[1][0] = genflg[2][1] = True;
    }
/*
 * Point group m-3m
 */

    else if (!strcmp(ptgrp,"m-3m"))
    {
        if (!strcmp(parts.r[0][1],"d") && !strcmp(parts.r[2][1],"m"))
        {
            wg[0][1].m[1][3] = 9;
            wg[0][1].m[2][3] = 6;
        }
        else if (!strcmp(parts.r[0][1],"d") && !strcmp(parts.r[2][1],"c"))
        {
            wg[0][1].m[0][3] = 9;
            wg[0][1].m[2][3] = 6;
        }
        if (!strcmp(parts.r[0][1],"a") && !strcmp(parts.r[2][1],"d"))
        {
            wg[0][1].m[0][3] = wg[0][1].m[2][3] = 6;
            wg[2][1].m[1][3] = wg[2][1].m[2][3] = 3;
        }
        else if (!strcmp(parts.r[2][1],"m"))
        {
            wg[2][1].m[0][3] = wg[0][1].m[0][3];
            wg[2][1].m[1][3] = wg[0][1].m[1][3];
            wg[2][1].m[2][3] = wg[0][1].m[2][3];
        }
        else
        {
            wg[2][1].m[0][3] = wg[0][1].m[0][3] + 6;
            wg[2][1].m[1][3] = wg[0][1].m[1][3] + 6;
            wg[2][1].m[2][3] = wg[0][1].m[2][3] + 6;
        }
```

```
            genflg[0][1] = genflg[1][0] = genflg[2][1] = True;
        }
        break;
/*
 * Rhombohedral and triclinic cells don't need any location adjustment
 */

    case 0: case 4:
/*
 * Mark any operator that is not an identity as a generator
 */

        for (i=0; i<4; i++)
        {
            for (j=0; j<2; j++)
            {
                for (x=0; x<3; x++)
                {
                    for (y=0; y<3; y++)
                    {
                        imat[x][y] = wg[i][j].m[x][y];
                    }
                }
                detntr(iptr, &det, &trace);
                genflg[i][j] = (trace != 3) ? True : False;
            }
        }
        break;
/*
 */
    default:
        break;
    }
/*
 * Once all of the possibilities have been handled, return the generators
 * flagged - note that the first generator is an identity matrix
 */

    k=1;
    for (i=0; i<4; i++)
    {
        for (j=0; j<2; j++)
        {
            if (genflg[i][j])
            {
```

```
                for (x=0; x<4; x++)
                {
                    for (y=0; y<3; y++)
                    {
                        gmats[k].m[x][y] = wg[i][j].m[x][y];
                    }
                    gmats[k].m[x][3] = wg[i][j].m[x][3] % 12;
                }
                k++;
            }
        }
    }
    ngen = k;
/*
 * Return the generators
 */
    return(ngen);
}
/*
 * toppri function
 */
#define I2D(ptr,x,y) *(ptr+2*x+y)
void toppri(int *pno, int *p1, int *p2, int *p3, int *p4)
{
/*
 * Returns the indices of the two lowest values in pno[0-2][0-1].
 * Ignores the inversion centre if it exists
 */

    int best, i, j;
    *p1=0;
    *p2=0;
    *p3=0;
    *p4=0;
    best=255;
    for (i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            if (I2D(pno,i,j) < best)
            {
                *p1 = i;
                *p2 = j;
                best = I2D(pno,i,j);
            }
```

```
        }
    }
    best=255;
    for (i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            if (!(i == *p1 && j == *p2) && I2D(pno,i,j) < best)
            {
                *p3 = i;
                *p4 = j;
                best = I2D(pno,i,j);
            }
        }
    }
    if (verb)
    {
        fprintf(stderr,"Best ops are %s & %s\n",parts.r[*p1][*p2], parts.r[*p3][*p4]);
    }
    return;
}
```

B.11 Spgpex Subroutine

```
/*
 * spgpex
 *
 * Common Blocks:
 */
#include <crack.h>
typedef struct _SPTRANS
{
    char *f, *t;
} SPTRANS;
static SPTRANS tr[] =
                    {
{"R32", "R 3 2 "},
{"H32", "H 3 2 "},
{"-6", "-6 "},
{"-4", "-4 "},
{"-3", "-3 "},
{"-1", "-1 "},
{"3121", "3_1 2 1"},
```

```
{"3112", "3_1 1 2"},
{"3221", "3_2 2 1"},
{"3212", "3_2 1 2"},
{"312", "3 1 2"},
{"321", "3 2 1"},
{"31m", "3 1 m"},
{"31c", "3 1 c"},
{"4322", "4_3 2 2"},
{"43212", "4_3 2_1 2"},
{"432", "4 3 2"},
{"4332","4_3 3 2"},
{"4232","4_2 3 2"},
{"4132","4_1 3 2"},
{"6222","6_2 2 2"},
{"622","6 2 2"},
{"42212","4_2 2_1 2"},
{"4222","4_2 2 2"},
{"422","4 2 2"},
{"4212","4 2_1 2"},
{"65","6_5 "},
{"64","6_4 "},
{"63","6_3 "},
{"62","6_2 "},
{"61","6_1 "},
{"41","4_1 "},
{"42","4_2 "},
{"43","4_3 "},
{"32","3_2 "},
{"31","3_1 "},
{"21","2_1 "},
{NULL,NULL}
};

void spgpex(char *in, char *out)
{
    char tmp[256];
    char *optr, *tptr, *rptr;
    int swap = False, i, j, k;

    optr = out;
    strcpy (tmp, in);

/*
 * if a rhombohedral symbol has an 'r' in it, set the lattice to 'R',
 * else if there is a 'h' or neither of these characters it defaults to 'H'.
```

```
*/

  for (rptr = NULL, tptr = tmp; *tptr; tptr++)
  {
      if (*tptr == 'R')
      {
          rptr = tptr;
          *rptr = 'H';
      }
      if (*tptr == 'r' && rptr)
      {
          *tptr = ' ';
          *rptr = 'R';
      }
      if (*tptr == 'h' && rptr)
      {
          *tptr = ' ';
          *rptr = 'H';
      }
  }

  for (tptr = tmp; *tptr; )
  {
      swap = False;
      for (i=0; *(tr[i].f); i++)
      {
          j=strlen(tr[i].f);
          if (!swap && !strncmp(tr[i].f, tptr, j))
          {
              tptr += j;
              j=strlen(tr[i].t);
              for (k=0; k<j; k++)
              {
                  *optr++ = tr[i].t[k];
              }
              swap = True;
          }
      }
      if (!swap && *tptr)
      {
          *optr++ = *tptr++;
      }
      *optr = NULL;
  }
  for (optr=out, tptr=tmp; *optr; optr++)
```

```
    {
        if (! isspace(*optr)) *tptr++ = *optr;
    }
    *tptr = NULL;
    strcpy (out, tmp);
    return;
}
```

B.12 Symeqs Subroutine

```
/*
 * symeqs
 *
 * Pete Ford, Durham University, June 1993
 * Permutes the existing symmetry operations to get a full set of equivalent
 * positions and checks for duplication.
 *
 * Common Blocks:
 */
#include <crack.h>
/*
 * Local variables:
 */
/*
 * Functions called:
 */
void pretty(int *, char *);
/*
 */

int symeqs(void)
{
    int trace, *tmat;
    int tnmats, g, h, i, j, k, n;
    LOGICAL exists;
/*
 * First copy the existing matrices into the large array
 */
    n = 0;
    for (k=0; k<11; k++)
    {
        trace = gmats[k].m[0][0] + gmats[k].m[1][1] + gmats[k].m[2][2];
        if ((trace < 3) || (k == 0))
```

```
        {
            for (i=0; i<4; i++)
            {
                for (j=0; j<4; j++)
                {
                    isym[n].m[i][j] = gmats[k].m[i][j];
                }
            }
            n++;
        }
    }
/*
 * Produce the string for each of the existing matrices; this will be
 * used to compare with new matrices
 */
    for (i=0; i<n; i++)
    {
        tmat = &(isym[i].m[0][0]);
        pretty (tmat,opstr[i].s);
    }
/*
 * Now try all the possible combinations of the existing matrices, AND all the
 * new ones produced
 */

    tnmats = n;
    for (g=0; g<tnmats; g++)
    {
        for (h=0; h<g+1; h++)
        {
/*
 * Multiply the matrices
 */
            for (j=0; j<4; j++)
            {
                for (i=0; i<4; i++)
                {
                    isym[n].m[i][j] = isym[g].m[0][j] * isym[h].m[i][0]
                                    + isym[g].m[1][j] * isym[h].m[i][1]
                                    + isym[g].m[2][j] * isym[h].m[i][2]
                                    + isym[g].m[3][j] * isym[h].m[i][3];
                }
            }
/*
 * Now calculate the string for the new matrix
```

```
 */
          tmat = &(isym[n].m[0][0]);
          pretty(tmat,opstr[n].s);
/*
 * Now compare this new string with the ones already known
 */
          exists = False;
          for (i=0; i<tnmats; i++)
          {
              if (!strcmp(opstr[n].s,opstr[i].s)) exists = True;
          }
/*
 * If the string does not exist, increment the counters since this must be a
 * new matrix
 */
          if (! exists)
          {
              n++;
              tnmats++;
              if (tnmats > 192)
              {
                  printf("symeqs: fatal error");
              }
          }
/*
 * Increment the counters and go to the top of the loop if not finished
 */
      }
   }
   return(tnmats);
}
```

B.13 Symget Subroutine

```
/*
 * syminit
 *
 * Reads the encoded symmetry matrices from  matrix2.h and converts the to
 * useable matrices
 *
 * Common Blocks:
 */
```

```
#include <crack.h>
#include <matrix2.h>
static int trans[] =
                  {
   -1,0,1,12,6,3,9,4,8,10,2
};

int symget(int op, int ax, int x, int y)
{
   char *cd;

   cd = symcodes[(op * 10 + ax) * 4 + x];
   return (trans[cd[y] - 'A']);
}
```

B.14 Crack Header File

```
#ifndef CRACK_H
#define CRACK_H
/*
 * Includes
 */

#include <stdio.h>
#include <stdlib.h>
/*
 * Defines
 */

#define LOGICAL int
#define True 1
#define False 0
#define Unknown -1
#define Labmax 10
#define Sgrpmax 30
/*
 * Structures and typedefs
 */

typedef struct MATRIX
{
   char f[4];
   int m[4][4];
```

```
} MATRIX;
typedef struct DMATRIX
{
   double m[4][4];
} DMATRIX;
typedef struct VECTOR
{
   double x, y, z;
} VECTOR;
typedef struct STRING
{
   char s[Labmax];
} STRING;

typedef struct SPBITS
{
   char f[80];
   char *r[3][2];
} SPBITS;
typedef struct SPSTRING
{
   char s[Sgrpmax];
} SPSTRING;
/*
 * Global variables
 */

SPSTRING    spgp,
gstring,
opstr[193],
class;
SPBITS      parts;
char        centring;
MATRIX      gmats[11],
matrix[11],
isym[193];
DMATRIX     symops[193];
int         nops;
LOGICAL     centric,
rgroup,
verb;
static int cvecs[5][3] =
                      {
{0,6,6},
{6,0,6},
```

```
{6,6,0},
{6,6,6},
{8,4,4},

};
#endif
```

B.15 Matrix Header File

```
/*
 *Pete Ford, Durham University, June 1993
 *Data file to be included in MATAX and ONEMAT, containing all of the operation
 *matrices for the 26 spacegroup operations, in the 10 forms for different
 *lattice vectors. Any case where the operation is not valid is left with a zero
 *matrix.
 */

extern MATRIX opmats[26][10];
/* String array containing symbols for operations */
 static char *ops[] =
 {
 "1","-1","2","m","21","a","b","c","n","d","3",
 "-3","31","32","4","-4","41","42","43","6","-6","61",
 "62","63","64","65",NULL
 };
 #define Identity 0
 #define Inversion 1
 #define Two_fold 2
 #define Mirror 3
 #define Two_one 4
 #define A_glide 5
 #define B_glide 6
 #define C_glide 7
 #define N-glide 8
 #define D_glide 9
 #define Three_fold 10
 #define Bar_three 11
 #define Three_one 12
 #define Three_two 13
 #define Four_fold 14
 #define Bar_four 15
 #define Four_one 16
 #define Four_two 17
```

```
#define Four_three 18
#define Six_fold 19
#define Bar_six 20
#define Six_one 21
#define Six_two 22
#define Six_three 23
#define Six_four 24
#define Six_five 25
/* String array with identifiers for crystal classes */
static char *cident[] =
{
"Tric","Mono","Orth","Trig",
"Tetr","Hexa","Cubi","Rhom",NULL
};
/* Axis numbers for the three axis symbols for each of the eight classes */
static int axisno[8][3] =
{
{1,1,1},
{0,1,2},
{0,1,2},
{2,5,4},
{2,0,3},
{2,8,9},
{2,6,3},
{6,7,0}
};
```

B.16 Matrix2 Header File

```
/*
 A = -1.0,
 B =  0.0,
 C =  1.0,
 D =  0.0 with variable flag,
 E =  0.5,
 F =  0.25,
 G =  0.75,
 H =  0.33333,
 I =  0.66667,
 J =  0.83333,
 K =  0.16667
*/
```

```
static char *symcodes[]=
                    {
/* identity */
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
 "BBBB","BBBB","BBBB","BBBB",
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
 "CBBD","BCBD","BBCD","BBBC",
/* inversion */
 "ABBB","BABB","BBAB","BBBC",
 "ABBB","BABB","BBAB","BBBC",
 "ABBB","BABB","BBAB","BBBC",
 "BBBB","BBBB","BBBB","BBBB",
 "BBBB","BBBB","BBBB","BBBB",
 "BBBB","BBBB","BBBB","BBBB",
 "BBBB","BBBB","BBBB","BBBB",
 "BBBB","BBBB","BBBB","BBBB",
 "BBBB","BBBB","BBBB","BBBB",
 "BBBB","BBBB","BBBB","BBBB",
/* 2 fold */
 "CBBB","BABD","BBAD","BBBC",
 "ABBD","BCBB","BBAD","BBBC",
 "ABBD","BABD","BBCB","BBBC",
 "BABB","ABBB","BBAD","BBBC",
 "BABB","ABBB","BBAD","BBBC",
 "CABB","BABB","BBAD","BBBC",
 "BBBB","BBBB","BBBB","BBBB",
 "BABD","ABBD","BBAD","BBBC",
 "CABB","BABB","BBAD","BBBC",
 "BABB","ABBB","BBAD","BBBC",
/* m plane */
 "ABBD","BCBB","BBCB","BBBC",
 "CBBB","BABD","BBCB","BBBC",
 "CBBB","BCBB","BBAD","BBBC",
 "BABD","ABBD","BBCB","BBBC",
 "BCBD","CBBD","BBCB","BBBC",
 "BABB","ABBB","BBCB","BBBC",
 "BBBB","BBBB","BBBB","BBBB",
 "BCBD","CBBD","BBCB","BBBC",
 "CBBB","CABB","BBCB","BBBC",
```

```
"ABBB","ACBB","BBCB","BBBC",
/* 21 screw */
"CBBE","BABD","BBAD","BBBC",
"ABBD","BCBE","BBAD","BBBC",
"ABBD","BABD","BBCE","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* a glide */
"BBBB","BBBB","BBBB","BBBB",
"CBBE","BABD","BBCB","BBBC",
"CBBE","BCBB","BBAD","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* b glide */
"ABBD","BCBE","BBCB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"CBBB","BCBE","BBAD","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* c glide */
"ABBD","BCBB","BBCE","BBBC",
"CBBB","BABD","BBCE","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BABD","ABBD","BBCE","BBBC",
"BCBD","CBBD","BBCE","BBBC",
"BABB","ABBB","BBCE","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BCBE","CBBE","BBCE","BBBC",
"CBBB","CABB","BBCE","BBBC",
"ABBB","ACBB","BBCE","BBBC",
```

```
/* n glide */
"ABBD","BCBE","BBCE","BBBC",
"CBBE","BABD","BBCE","BBBC",
"CBBE","BCBE","BBAD","BBBC",
"BABE","ABBE","BBCE","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* d glide */
"ABBD","BCBF","BBCF","BBBC",
"CBBF","BABD","BBCF","BBBC",
"CBBF","BCBF","BBAD","BBBC",
"BCBF","CBBF","BBCF","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 3 fold */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BABD","CABD","BBCB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBCB","CBBB","BCBB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* -3 rotoinversion */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BCBD","ACBD","BBAD","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBAB","ABBB","BABB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 31 screw */
```

```
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BABD","CABD","BBCH","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 32 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BABD","CABD","BBCI","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 4 fold */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BABD","CBBB","BBCB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* -4 rotoinversion */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BCBD","ABBB","BBAD","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 41 screw */
"BBBB","BBBB","BBBB","BBBB",
```

```
"BBBB","BBBB","BBBB","BBBB",
"BABD","CBBB","BBCF","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 42 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BABD","CBBB","BBCE","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 43 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BABD","CBBB","BBCG","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 6 fold */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"CABD","CBBD","BBCB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* -6 rotoinversion */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
```

```
"ACBD","ABBD","BBAB","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 61 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"CABD","CBBD","BBCK","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 62 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"CABD","CBBD","BBCH","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 63 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"CABD","CBBD","BBCE","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 64 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"CABD","CBBD","BBCI","BBBC",
```

```
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
/* 65 screw */
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"CABD","CBBD","BBCJ","BBBC",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
"BBBB","BBBB","BBBB","BBBB",
NULL
};
```