dkfz

Department
MBI

MBI
Technical Report
TR 73/95

The guide to efficient use and programming with PIC images

Authors:	A.Schroeder, U. Baur, U. Engelmann

Document Number	TR-73/95
Document Version	1.0
Date	Juli 1995
Operating Systems and Versions:	Without Restrictions

This document gives an introduction for the work with PIC images at the German Cancer Research Center in Heidelberg, Germany

Deutsches Krebsforschungszentrum

Im Neuenheimer Feld 280, D-69120 Heidelberg, Germany

The information in this document is subject to change without notice.The department MBI of the German Cancer Research Center in Heidelberg, Germany, assumes no responsibility for any errors that may appear in this document.

DEC, DECstation, DECsystem, ULTRIX and VT are registered trademarks of Digital Equipment Corporation.

FrameMaker is a trademark of Frame Technology Corporation.

OSF, OSF/1, OSF/Motif and Motif are trademarks of the Open Software Foundation.

SUN is a registered trademark of SUN Microsystems, Inc.

Unix is a registered trademark of UNIX System Laboratories, Inc.

Preface

Objectives of the manual

This guide explains how the PIC format is used efficiently and emphasises the construction of compact, easy to use and fast image processing functions in the programming language C.

Audience

This document is written for programmers who have to start working with PIC image format.

Document structure

This document covers the following topics:

- Chapter 1: Introduction
- Chapter 2: PIC format
- Chapter 3: Investigation of different programming alternatives
- Appendix A: Program listings

Related documents

The documents most likely to help you getting started are:

- Technical Report: PIC Image File Standard. U.Engelmann, A.Schröter
- Technical Report: PIC C Library. A. Schröter
- Technical Report 56/94: MBI C Coding Standards. A. Schröter, U. Günnel, U. Engelmann

Conventions

Convention	Meaning
times	The default character type used in all documents
`courier`	Words and letters in courier type font in examples should be typed as is. This font is used for programming examples.
helvetica	Syntax descriptions (BNF forms) are described with helvetica type characters (11 pt).
boldface text	Times 11 pt boldface text is used to show user input during interactive dialogues.
italics	Times italic type indicates variable values, placeholders, and function arguments names.
[]	In format descriptions, brackets indicates that whatever is enclosed within the brackets is optional.
{}	In format descriptions, braces surround a required choice of options of which one and only one must be selected.
|	A vertical bar separates elements of which one should be selected.
…	In examples, a horizontal ellipsis indicates that the preceding items or items can be repeated. If the ellipsis is preceded by a comma, the repeated elements should be separated by a comma.
[ ]…	Indicates that the element within brackets can be repeated from 0 to *n* times.
{ }…	Indicates that the element within braces can be repeated from 1 to *n* times.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format.
::=	The notation ::= indicates that the following elements belong to the definition.
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labelled Ctrl while you press another key or a pointing device button.

Contents

1 Introduction

This document explains how the PIC format is used efficiently. It emphasises the construction of compact, easy to use and fast image processing functions in the programming language C.

The purpose of this report is to be a help for everybody who has to start working with this image format. On one hand the general structure of an image which ist stored in this format is described and on the other hand some hints which could be usefull for programming are given.

After having given a short overview of the structure of the PIC format in chapter 2.2 the most important functions for handling an image (e.g. for reading, saving an image or allocating memory for an image) are explained shortly in chapter 2.3.1. In this chapter it is also shown, how the information of a single pixel is stored and how this data could be accessed.

After this general information the investigation of different programming alternatives is described in chapter 3. The development of two programs (the first is used to invert the greyvalues of an image and the second to reflect an image along one axis) is explained quite detailed.

2 PIC image format

This chapter recalls the PIC image format shortly. For more details see []. After some general remarks the structure of the PIC image is described and some programmin hints are given.

2.1 General remarks

The programming language C has been chosen for the implementation of the PIC format. To compile and link own functions which handle images encoded in PIC format some include files and libraries have to be inserted The include files which are needed are locally stored in the directory `/usr/local/include` and subsequent subdirectories. The most important include files are

- `ipPic.h` in subdirectory `ipPic.` This include file contains all the necessary structures (e.g `ipPicDescriptor` for the image structure) and the macros which are explained in chapter 3.
- `ipTypes.h` in subdirectory `ipPic`. This include file contains the definition of the used datatypes. According to the MBI C Coding Standards[] these data-types should be used to be independent from the hardware the programs are running on.
- `ipFunc.h` in subdirectory `ip`. This include file contains the prototypes of some functions to transform images.

The libraries which are needed are in directory `/usr/local/lib`. The most important libraries are

- `libipPic.a` This library contains functions to handle images like read, write or copy images.
- `libipFunc.a`. This library contains the functions to transform images.

2.2 Structure of the PIC image format

An image stored in PIC-format consist of three parts

- header

- additional information
- pixel data

2.2.1 Header

The pixel data are described in the header. That means that it contains information about the data-type of the pixels, the length of a pixel in storage, the number of dimensions and the size of each dimension. At the beginning there is an identification string which contains information about the PIC format and the version.

There are about eight data-types. The most common of them are:

type	**machine independent types**	**code**
integer	ipPicInt	3
unsigned integer	ipPicUInt	4
float	ipPicFloat	5

Figure 2–1 possible datatypes

The information of the second and the third row are equivalent, because the type is defined as an enumeration type. That means that each type in this list is attached to a number.

2.2.2 Additional information in the header

The tag fields contain additional information belonging to the image. This additional information could for example be a look-up-table which should be used to display the picture. Another example is a greylevel window which should be seen by default when the picture is figured. All tag-fields are collected in a Tag-Length-Structure-Value (TLSV) - quadrupel. For detailed information about these TLSV-quadrupels see [???].

2.2.3 Pixel data

The pixel data are stored sequential without any delimiters at the very end of the file. That means that the stored data don’t depend on the type of the pixels. To access the data it is necessary to extract the type out of the information of the header. This will be described in detail in chapter 2.4.

2.2.4 Representation of the PIC image format in C

In the programming language C an image in PIC format is described by a structure called ipPicDescriptor, which is shown in the following lines

```
typedef struct
 {
   void       *data;      /* pointer to image data    */
```

```
    _ipPicInfo_t *info;      /* pointer to PicInfo
*/
                              /* structure             */
    ipPicType    type;      /* data type of data     */
    ipUInt4_t    bpe;       /* bits per element      */
    ipUInt4_t    dim;       /* number of dimensions  */
    ipUInt4_t    n[_ipPicNDIM]
                           /* size of dimension n[i]*/
  } ipPicDescriptor
```

The meaning of the single components of this structure was described in chapter 2.2. The data are described in chapter 2.2.3. The type, bpe, dim and n[_ipPicNDIM] components belong to the header, which is described in chapter 2.2.1. The aditional information, which are represented by the component `info` of the ipPicDescriptor, is explained in chapter 2.2.2. This component is of type `_ipPicInfo_t`

```
typedef struct
  {
          ipPicTag_t  version;
   _ipPicTagsElement_t *tags_head;
           ipBool_t  write_protect;
  } _ipPicInfo_t;
```

The component `tags_heads` contains nearly the same components as the header.

2.3 Programming hints

2.3.1 Functions for image I/O and manipulation

There are quite a lot of functions to handle an image. To access these functions the library libipPic.a has to be used.

Some of these functions which are necessary to start working with the PIC format are described in this chapter. This list should only be a short synopsis. For more detailled information see [???].

All functions which use the described function have to start with the following declaration.

```
#include <ipPic/ipPic.h>

ipPicDescriptor *pic, *pic_old, *pic_new;
       /* pointer to the image structures                */
char            file_name[NAME_MAX];
       /* name of the image file                         */
ipUInt4_t       size, elements;
```

pic = ipPicGet (file_name, pic_old)

This function reads an image from disk into the returned ipPicDescriptor. If the parameter pic_old is NULL before executing this function the memory which is needed is allocated otherwise it is reused.

ipPicPut (file_name, pic)

This function saves an image belonging to the ipPicDescriptor pic in a file named file_name.

pic = ipPicNew( )

Returns a pointer to the image structure and is used to allocate this image structure. Because there is no memory for the pixel data allocated, this still has to be done. The information of the header is needed to allocate this memory.

pic_new = ipPicCopyHeader (pic_old, pic_new)

This function copies the header of an ipPicDescriptor pic_old to the header of pic_new. A new structure is allocated if pic_new is NULL before executing this function . Otherwise the program reuses the existing one. It is also necessary to allocate memory for the image data.

size = _ipPicSize (pic)

This function calculates the size of the image data of pic in bytes by using the information of the header and returns it.

elements = _ipPicElements (pic)

This function calculates the number of pixels in the image based on the information of the header and returns it.

ipPicClear (pic)

This function clears an ipPicDescriptor. All memory allocated for the image data of pic is freed.

In addition to these functions there are also some macros and constants implemented which could be useful when acessing and manipulating the image data.

_ipPicNDIM is a constant which contains the maximal number of dimensions of a picture (in PIC format it is eight at the moment).

ipPicDR is a macro that combines the type and bpe (bit per element) of the image data to the Data Representation Code, which makes it easier to handle pictures from different image data types.

2.4 Accessing the image data

As mentioned before the image data are stored bytewise and sequential independent of the datatype. If this data should be manipulated in any way it is necessary to know their type and accessing them by using this type. That means that before using the data a typecast is necessary. To explain how to do this it could be useful to look at the following figure which shows the image data in memory.

Depending on the length of a pixel (see pic->bpe = bits per element) the corresponding number of bytes is interpreted as one pixel.

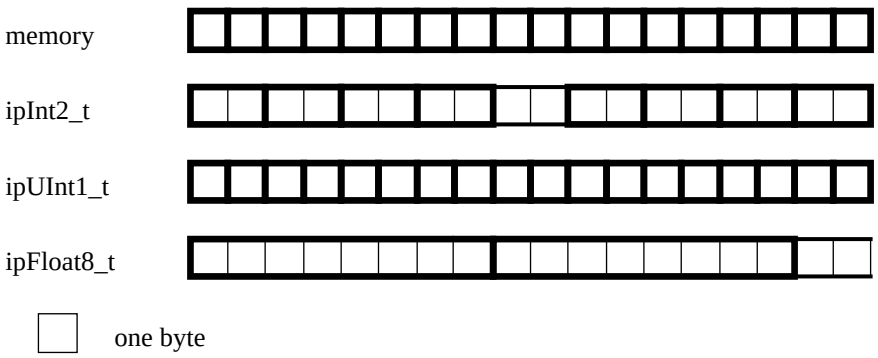

Figure 2–2 image data in memory

The type is important for the internal representaion of the data. Because the names of the different data types depend on the operating system the user defined datatypes of MBI (which are defined in the include file ipPic/ipTypes.h) should be used. Following data types are possible

- ipInt1_t : signed integer number, stored in one byte
- ipInt2_t : signed integer number, stored in two bytes
- ipInt4_t : signed integer number, stored in four bytes
- ipUInt1_t : unsigned integer number, stored in one byte
- ipUInt2_t : unsigned integer number, stored in two bytes
- ipUInt4_t : unsigned integer number, stored in four bytes
- ipFloat4_t : float number, stored in four bytes
- ipFloat8_t : float number, stored in eight bytes

To find out the data type the macro `ipPicDR ( pic->type, pic->bpe )` could be taken. It has to be used in the following way

```
switch ( ipPicDR ( pic->type, pic->bpe ) )
{
  case ( ipPicDR ( ipPicInt, 8 ) ) :
    /*  accessing ipInt1_t data  */
    break;
  case ( ipPicDR ( ipPicInt, 16 ) ) :
    /*  accessing ipInt2_t data  */
    break;
  case ( ipPicDR ( ipPicInt, 32 ) ) :
```

```
      /*  accessing ipInt4_t data  */
      break;
    case ( ipPicDR ( ipPicUInt, 8 ) ) :
      /*  accessing ipUInt1_t data */
      break;
    case ( ipPicDR ( ipPicUInt, 16 ) ) :
      /*  accessing ipUInt2_t data  */
      break;
    case ( ipPicDR ( ipPicUInt, 32 ) ) :
      /*  accessing ipUInt4_t data  */
      break;
    case ( ipPicDR ( ipPicFloat, 32 ) ) :
      /*  accessing ipFloat4_t data */
      break;
    case ( ipPicDR ( ipPicFloat, 64 ) ) :
      /*  accessing ipFloat8_t data */
      break;
    default :
      printf ( " datatype doesn’t exist \n " );
  }
```

After having determined the datatype of the image data by using the number of bits per element and the pic type the typecasting has to be done. For example a picture of type ipPicFloat and 64 bit per element (pixel) is internally represented by an ipFloat8_t number. Acessing data could be done in the following way. It is shown in an example which copies the greyvalues of one image to another.

```
for ( i = 0; i < _ipPicElements ( pic_old ); i++ )
  ( ( ipFloat8_t * ) pic_new->data ) [i] = ( ( ipFloat8_t
* ) pic_old->data )
```

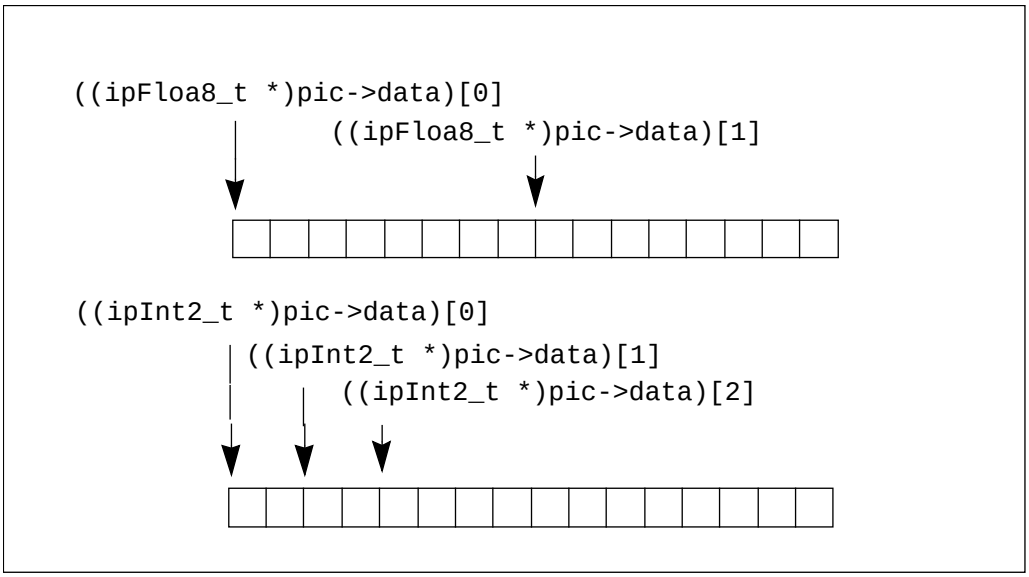

Figure 2–3 Accessing image data of different image data type

[Pictures could be multidimensional. Animage with n dimensions is stored in an n-dimensional array. For example a sequence of three dimensional pictures is stored in a four dimensional array. The size of the first dimension corresponds to the number of rows, the second to the number of lines, the third to the number of plains and the fourth to the number of pictures in a sequence. The order in which the pixels are stored in memory corresponds to this order. To access pixel j in line i in the k^{th}-plain of the l^{th}- image the following syntax can be used

```
pic->data [ j + i*pic->n[0] + k*pic->n[0]*pic->n[1] +
          l*pic->n[0]*pic->n[1]*pic->n[2] ]
```

or

```
pic->data [j] [i] [k] [l]
```

where pic->n[m] is the size of the corresponding dimension.

2.5 Further information

Some information about an image can be obtained by the program picinfo, which is started on the operation system level by

```
picinfo filename [-e] [-h]
```

When adding the option -e the maximum and minimum greyvalues are printed additionally. Adding the option -h some help about the usage of picinfo is given.

cshow or movie could be used to display PIC images. They can be started by

```
cs  filename
movie filename
```

cs is taken to show two dimensional images and movie for three dimensional image or image sequences. With cs -h or movie -h all commandline parameters are listed.

3 Investigation of different programming alternatives

In image processing different classes of operators are differentiated. The most common classes are

- Operators to transform greyvalue: The greyvalue of each pixel is changed independent of of the surrounding pixels (e.g. the inversion of the greyvalues).
- Local operators: Thgreyvalues of the surrounding pixels are used to calculate the greyvalue of the pixel which is changed (e.g. filters).
- Global operators: All pixels of the image are used to calculate the new greyvalue of the current pixel.
- Operators which regard the geometry of the image (e.g reflection of an image).

In this chapter the development of two programs is illustrated. The first one is a program to invert the greyvalues of an image. This is an operation which is not depending on the position of the single pixels that should be changed. It belongs to the first class of operators described above. The second program reflects an image along one axis which could be entered by the user. This operation belongs to the class of operators which regard the geometry of the image.

In the first version of both programs the type of the image data is determined and afterwards branched to the corresponding code. That effects very long code because the whole operation has to be coded for each possible datatype. The only difference in these code segments is the typecast for the pixel data. For this reason in later versions macros are introduced to replace the switch instruction to find out the data type.

In the reflection program there are some other steps necessary because the operation depends on the pixel position, the number of dimensions must be regarded and the axis along which the reflection has to be done also. These steps are descibed in the second part of this chapter.

3.1 Function invert

3.1.1 Description of the program (without using macros)

The original picture is read by using the function `ipPicGet` in the following way

```
pic = ipPicGet ( pic_name, NULL )
```

where pic is of type `ipPicDescriptor` and is a pointer to the image structure in memory, pic_name is the name of the file in which the image is stored. NULL means that an `ipPicDescriptor` has to be allocated. After having read the picture it is inverted in the function picInv. For the inversion the following algorithm is used:

```
result_greyvalue = max + min - greyvalue
```

It effects that the greatest occuring greyvalue is assigned to the lowest and vice versa. In function `picInv` a new `ipPicDescriptor` for the inverted image is allocated and initialized. This is done by using the function ipPicCopyHeader. Then memory for the inverted image is allocated.

```
pic_new = ipPicCopyHeader ( pic, NULL )
pic_new->data = malloc ( _ipPicSize ( pic_new ) )
```

After that the type of the image data is found out by using the macro `ipPicDR` and the following switch statement

```
switch ( ipPicDR ( pic->type, picpic_old->bpe ) )
{
  case ( ipPicDR ( ipPicInt, 8 ) ) :
    /* inverting an ipInt1_t image   */
    break;
  case ( ipPicDR ( ipPicInt, 16 ) ) :
    /* inverting an ipInt2_t image   */
    break;
  case ( ipPicDR ( ipPicInt, 32 ) ) :
    /* inverting an ipInt4_t image   */
    break;
  case ( ipPicDR ( ipPicUInt, 8 ) ) :
    /* inverting an ipUInt1_t image  */
    break;
  case ( ipPicDR ( ipPicUInt, 16 ) ) :
    /* inverting an ipUInt2_t image  */
    break;
  case ( ipPicDR ( ipPicUInt, 32 ) ) :
    /* inverting an ipUInt4_t image  */
    break;
  case ( ipPicDR ( ipPicFloat, 32 ) ) :
    /* inverting an ipFloat4_t image  */
    break;
```

```
    case ( ipPicDR ( ipPicFloat, 64 ) ) :
      /* inverting an ipFloat8_t image */
      break;
    default :
      printf ( " datatype doesn't exist \n " );
  }
```

In the following lines the code for the inversion is shown for an image of type `ipInt2_t`

```
  ipInt2_t help;

  /* calculate the maximum an minimum greyvalue of the    */
  /* image                                                */

  ipInt2_t max = ( ( ipInt2_t * ) pic->data ) [0];
  ipInt2_t min = ( ( ipInt2_t * ) pic->data ) [0];

  for ( i = 1; i < _ipPicElements ( pic ); i++ )
  {
    help =( ( ipInt2_t * ) pic->data ) [i]
    max = ( help > max ) ? help : max;
    min = ( help < min ) ? help : min;
  }

  /* inverting the greyvalues                             */

  for ( i = 0; i < _ipPicElements ( pic ); i++ )
    ( ( ipInt2_t * ) pic_new->data ) [i] =
        max + min - ( ( ipInt2_t * )pic->data ) [i];
```

A construction like

```
  max = ( help > max ) ? help : max;
```

is used for calculating the extreme values. In this statement first the boolean expression is evaluated. Depending on the result `max` is assigned the variable `help` (in case the result of the comparison is true) or the variable `max` (in case the result of the comparison is false).

3.1.2 Description of the program (using macros)

As seen in chapter 3.1.1 the code gets very long when checking each possible datatype. For making the code clearer, macros are introduced. These macros are equivalent to the switch statement mentioned in the last chapter. When compiling the program the preprocessor expands this macro. That effects the same code as if the type is explicitly checked.

There is no difference in the main programs of this two versions but in the function picInv. Instead of the switch statement a macro is used. This macro is called `ipPicFORALL` and has two parameters. The first one is a macro again which contains the source code for inverting the picture. The second parameter

is the ipPicDescriptor of the image which should be inverted. The macro `ipPicFORALL` already exists and is defined in the include file `ipPic/ipPic.h` in directory `/usr/local/include`.

Because of using this macro the source code of the function `picInv` gets rather short. It's just necessary to copy the header of the old picture and to allocate the memory before using the macro.

```
pic_new = ipPicCopyHeader ( pic_old, NULL );
pic_new->data = malloc ( _ipPicSize ( pic_new ) );

ipPicFORALL ( INV, pic_old );
```

INV is also a macro which is defined before the function `main`. This macro has also two parameters. The first is the type of the picture which is determined in the macro `ipPicFORALL`. This parameter type is replaced by the real type when the macro is expanded. The second parameter is the ipPicDescriptor for the image which should be inverted. The algorithm is the same as described in the first program . It is important that there is no blanc between the macro name and the bracket in the define statement and that a macro has to be coded in one line. Because of that each line has to end with a "\".

```
/* definition of invert macro                                    */

#define INV( type, pic )                                          \
{                                                                 \
  ipUInt4_t  i;                                                   \
  type      help;                                                 \
                                                                  \
  type  max = ( ( type * ) pic->data )[0];                        \
  type  min = ( ( type * ) pic->data )[0];                        \
                                                                  \
  for ( i = 1; i < _ipPicElements ( pic ); i++ )                  \
     {                                                            \
       help = ( ( type * ) pic->data ) [i];                       \
       max = ( help > max ) ? help : max;                         \
       min = ( help < min ) ? help : min;                         \
     }                                                            \
                                                                  \
  for ( i = 0; i < _ipPicElements ( pic ); i++ )                  \
     {                                                            \
       (( type * ) pic_new->data ) [i] =                          \
           max + min - (( type * ) pic->data )[i];                \
     }                                                            \
}
```

There are five different kinds of the macro `ipPicFORALL`. They are called `ipPicFORALL, ipPicFORALL_1, ..., ipPicFORALL_4`. The number at the

end of the name tells the user how many parameters the macro (which is the first parameter of the `ipPicFORALL` macro) could have additionally.

For example if the extreme values are calculated before the use of the macro the macro would be

```
ipPicFORALL_2 ( INV, pic_old, max, min );
```

The define statement for this macro would be then

```
#define INV( type, pic, max, min )                          \
{                                                           \
  ipUInt4_t  i;                                             \
                                                            \
   for ( i = 0; i < _ipPicElements ( pic ); i++ )           \
     {                                                      \
       (( type * ) pic_new->data ) [i] =                    \
         max + min - (( type * ) pic->data )[i];            \
     }                                                      \
}
```

In this case also a macro for calculating the extreme values has to be written. Because the extreme values are needed in some other functions it is quite useful to have such a macro. A solution which uses this macro is used in libipFunc.a.

3.2 Function reflect

When reflecting an image the position of the pixels is important because there are always two pixels which are exchanged. For example if a two dimensional image should be reflected along the y-axis the first and the last pixel in a line have to be exchanged. Generally, in each line pixel i and pixel (n - i - 1) have to be exchanged. In this case n is the size of the first dimesion.

For a n-dimensional image reflection along n axis is possible.When the reflection is programmed without using macros the code gets very long because the algorithm has to be implemented for each data type along each axis. This way is realized in the first version of the program described in chapter 3.2.1. In the second version the algorithm is generalized. It could be used for reflection along each axis (see chapter 3.2.2). In the next step a macro is introduced to generalize the program for each possible data type (see chapter 3.2.3). At least the macro is changed in that way that it could be used for each problem in which the operation is depending on the pixel position.

3.2.1 First version - detailed reflection

This version of the program is rather long because the algorithm has to be coded for each data type separatly. The algorithm also differs depending on the axis along which the reflection has to be executed. Considering just four dimensional images the reflection could be described as follows

reflection axis	**algorithm**
x	p (j, i, k, l) = p (j, n[1]-i-1, k, l)
y	p (j, i, k, l) = p (n[0]-j-1, i, k, l)
z	p (j, i, k, l) = p (j, i, n[2]-k-1, l)
t	p (j, i, k, l) = p (j, i, k, n[3]-l-1)

Figure 3–1 algorithm for reflection

where p (j, i, k, l) is the pixel j in line i of plain k in the picture l of the sequence and n[m] the number of elements in the corresponding direction. It is necessary to subtract one from n[m] because in C an array starts with index zero.

The first version of this program could only be used for four dimensional images. The general structure is quite similar to the structure of the programs in chapter 3.1 . After having read the image the reflection is done in function picRefl. In this function a copy of `pic_old` is made and memory allocated first. The reflection is done in dependence of the type and the axis in a switch statement (see chapter 3.1.1). The reflection is coded as followed

```
for ( i = 0; i < pic_old->dim; i++ )
   n[i] = pic_old->n[i];

for ( i = pic_old->dim; i < _ipPicNDIM; i++ )
   n[i] = 1;

/* y-axis                                          */

if ( achse == 2 )
{
   for ( l = 0; l < n[3]; l++ )
   {
     z1 = l * n[2] * n[1] * n[0];
     for ( k = 0; k < n[2]; k++ )
     {
       z2 = k * n[1] * n[0];
       for ( i = 0; i < n[1]; i++ )
       {
         z3 = i * n[0];
         for ( j = 0; j < n[0]; j++ )
         {
            (( ipInt1_t * )pic_new->data )[j+z1+z2+z3] =
                (( ipInt1_t * )pic_old->data )
                                    [n[0]-j-1+z1+z2+z3];
         }
       }
     }
   }
 }
```

```
/* x-axis                                                */

if (  achse == 1 )
{
  for ( l = 0; l < n[3]; l++ )
  {
    z1 = l * n[2] * n[1] * n[0];
    for ( k = 0; k < n[2]; k++ )
    {
      z2 = k * n[1] * n[0];
      for ( i = 0; i < n[1]; i++ )
      {
        z3 = i * n[0];
        z4 = ( n[1] - i - 1 ) * n[0];
        for ( j = 0; j < n[0]; j++ )
        {
           (( ipInt1_t * )pic_new->data )[j+z1+z2+z3] =
               (( ipInt1_t * )pic_old->data )
                                    [j + z1 + z2 + z4];
        }
      }
    }
  }
}

/* z axis                                                */

if ( achse == 3 )
{
  for ( l = 0; l < n[3]; l++ )
  {
    z1 = l * n[2] * n[1] * n[0];
    for ( k = 0; k < n[2]; k++ )
    {
      z2 = k * n[1] * n[0];
      z3 = ( n[2] - k - 1 ) * n[1] * n[0];
      for ( i = 0; i < n[1]; i++ )
      {
        z4 = i * n[0];
        for ( j = 0; j < n[0]; j++ )
        {
           (( ipInt1_t * )pic_new->data )[j+z1+z2+z4] =
             (( ipInt1_t * )pic_old->data )
                                    [j + z1 + z3 + z4];
        }
      }
    }
  }
```

```
 }

/* t-axis                                                */

if ( achse == 4 )
{
   for ( l = 0; l < n[3]; l++ )
   {
     z1 = l * n[2] * n[1] * n[0];
     z2 = ( n[3] - l - 1 ) * n[2] * n[1] * n[0];
     for ( k = 0; k < n[2]; k++ )
     {
       z3 = k * n[1] * n[0];
       for ( i = 0; i < n[1]; i++ )
       {
         z4 = i * n[0];
         for ( j = 0; j < n[0]; j++ )
         {
            (( ipInt1_t * )pic_new->data )[j+z1+z3+z4] =
              (( ipInt1_t * )pic_old->data )
                                    [j + z2 + z3 + z4];
         }
       }
     }
   }
 }
```

where z1 upto z4 are used to save cpu-time because these values otherwise have to be calculated in each step of the loop. The vector n contains the number of pixels in each direction. In case that images with less than four dimensions should be reflected, the components of the vector n which belong to those directions have to be initialized to one. That effects that the loops which belong to those directions are executed only once.

Because the code is very long the next step is to generalize the algorithm that it suits for all axis. This algorithm will be explained in the next chapter.

3.2.2 Second version - Reflection along every axis

In this chapter it is described, how the algorithm for the reflection could be generalized. The algorithm is also extended to the number of dimensions which is at most possible (eight in PIC-format).

To generalize the algorithm it could be useful to look carefully at it. The pixel of the reflected image could be calculated as follows

```
pic_new->data [j + i*n[0] + k*n[0]*n[1] +
               l*n[0]*n[1]*n[2]] = pic->data [offset]
```

where `offset` is (depending on the reflection axis).

axis	**offset**			
y	(n[0]-j-1)+	i*n[0] +	k*n[0]*n[1] +	l *n[0]*n[1]*n[2]
x	j +	(n[1]-i-1)*n[0] +	k*n[0]*n[1] +	l *n[0]*n[1]*n[2]
z	j +	i*n[0] +	(n[2]-k-1)*n[0]*n[1] +	l *n[0]*n[1]*n[2]
t	j +	i*n[0] +	k*n[0]*n[1] +	(n[3]-l-1) *n[0]*n[1]*n[2]

Figure 3–2 algorithm depending on the axis

In this table could be seen that the only position where the algorithm differs is the diagonal element. Because the factors n[0] * ... * n[k] are not depending on the loop index they could be calculated before those loops. In the pogram a vector called `length_vect` is used for this factors.

```
length_vect[0] = 1;
for ( i = 1; i < _ipPicNDIM; i++ )
  length_vect[i] = length_vect[i-1] * n[i];
```

There is also a vector called `axis_vect`, which contains zero in all elements except the element corresponding to the axis along which the reflection should be done. This elements gets the number of pixels in that direction.

```
for ( i = 1: i < _ipPicNDIM; i++ )
  axis_vect[i] = 0];
axis_vect[achse - 1] = n[achse - 1] - 1;
```

At least there is one vector which is called `index_vect`, which is used for the counting variables in the loop. It’s initialized to zero.

Using these vectors the reflection could be described as follows

axis	**0**	**1**	**2**	**3**
y	(axis_vect[0]-index_vect[0]) * length_vect[0] +	index_vect[1] * length_vect[1] +	index_vect[2] * length_vect[2] +	index_vect[3] * length_vect[3] +
x	index_vect[0] * length_vect[0] +	(axis_vect[1]-index_vect[1]) * length_vect[1] +	index_vect[2] * length_vect[2] +	index_vect[3] * length_vect[3] +
z	index_vect[0] * length_vect[0] +	index_vect[1] * length_vect[1] +	(axis_vect[2]-index_vect[2]) * length_vect[2] +	index_vect[3] * length_vect[3] +
t	index_vect[0] * length_vect[0] +	index_vect[1] * length_vect[1] +	index_vect[2] * length_vect[2] +	(axis_vect[3]-index_vect[3]) * length_vect[3] +

Figure 3–3 algorithm (after introducing the vectors)

As mentioned before the algorithm for the different axis just differs in the diagonal elements. To calculate the offset of the pixels which have to be exchanged with the current one the four components have to be added. If axis_vect[i] is not zero, the exprssion `(axis_vekt[i] - index_vekt[i]) * length_vekt[i]` is added, otherwise `index_vekt[k] * length_vekt[k]` is added. To check whether `axis_vekt[i]` is zero the same construction (() ? () : ()) as described in chapter 3.1.1 to calculate the extreme values is used.

Because the pixels of the reflected image are transformed in the order which they are stored in, the offset (`offset_refl`) has just to be incremented.

The algorithm then is (shown for an image of type `ipInt1_t`)

```
offset_refl = 0;
for ( index_vect[7] = 0; index_vect[7] < n[7];
                          index_vect[7]++ )

 for ( index_vect[6] = 0; index_vect[6] < n[6];
                          index_vect[6]++ )

  for ( index_vect[5] = 0; index_vect[5] < n[5];
                           index_vect[5]++ )

   for ( index_vect[4] = 0; index_vect[4] < n[4];
                            index_vect[4]++ )

    for ( index_vect[3] = 0; index_vect[3] < n[3];
                             index_vect[3]++ )

     for ( index_vect[2] = 0; index_vect[2]<n[2];
                              index_vect[2]++ )

      for ( index_vect[1] = 0; index_vect[1] < n[1];
                               index_vect[1]++ )

       for ( index_vect[0] = 0; index_vect[0] < n[0];
                                index_vect[0]++)
       {
       offset_orig = 0;
       for ( i = 0; i < pic_old->dim; i++ )
       {
          offset_orig += length_vect[i] *
             (( axis_vect[i] == 0 ) ?
             index_vect[i] :
             ( axis_vect[i]-index_vect[i] ));
       }
       ((ipInt1_t *)pic_new>data)[offset_refl] =
          (( ipInt1_t * ) pic_old->data  [offset_orig];
       offset_refl++;
      }
```

The only problem in this program is that the algorithm has to be implemented for each possible data type. This problem is solved by using macros as shown in the next chapter.

3.2.3 Third version - type independent reflection

In this program the switch statement which was used to find out the type of the image data and to do the reflection is replaced by the call for the macro ipPicFORALL_4 which is one of those macros that were explained in chapter 3.1.2. It is a macro which could be used for functions with four parameters. The first parameter of the ipPicFORALL_4 macro again is a name of a macro. This macro contains the algorithm for the reflection. The second parameter is the picture that should be reflected. The last four parameters are needed in the reflection macro

```
ipPicFORALL_4 ( REFL, pic_old, axis_vect, length_vect,
             index_vect, n );
```

The macro REFL looks as follows

```
#define REFL( type, pic, axis, length, index,n)          \
{                                                        \
  ipUInt4_t   offset_orig;                               \
  ipUInt4_t   offset_refl;                               \
                                                         \
  offset_refl = 0;                                       \
  for ( index[7] = 0; index[7] < n[7]; index[7]++ )      \
                                                         \
    for ( index[6] = 0; index[6] < n[6]; index[6]++ )    \
                                                         \
      for ( index[5] = 0; index[5] < n[5]; index[5]++ )  \
                                                         \
        for ( index[4] = 0; index[4] < n[4]              \
                            index[4]++ )                 \
          for ( index[3] = 0; index[3] < n[3];           \
                              index[3]++ )               \
            for ( index[2] = 0; index[2] < n[2];         \
                                index[2]++ )             \
              for ( index[1] = 0; index[1] < n[1];       \
                                  index[1]++ )           \
                for ( index[0] = 0; index[0] < n[0];     \
                                    index[0]++ )         \
                {                                        \
                  offset_orig = 0;                       \
                  for ( i = 0; i < pic->dim; i++ )       \
                  {                                      \
                    offset_orig += length[i] *           \
                      (( axis[i] == 0 ) ?                \
                        index[i] : ( axis[i]-index[i]));  \
                  }                                      \
```

```
                    (( type * )pic_new->data) [offset_refl]=\
                        (( type * )pic->data) [offset_orig]; \
                    offset_refl++;                           \
                  }                                          \
}
```

After these macros are expanded the code is corresponding with that one which was described in chapter 3.2.2.

3.2.4 Fourth version - minimized, generalized reflection

The code gets quite short when using the macros as described in chapter 3.2.3. Because there are some more operations in image processing which depend on the position of the pixels so that they need this eight nested loops, it would be nice to have a macro which replaces these loops. In the call for the macro only the algorithm for the wanted operation has to be inserted.

The first change is the call for the macro `ipPicFORALL_4`. The first parameter is called `FORLOOP` and is a macro which contains the eight nested loops. The last parameter contains the operation which schould be applied to the image. It calculates the offset of the pixel which should be exchanged with the current one.

```
ipPicFORALL_4 ( FORLOOP, pic_old, pic_new,
                index_vect, offset_orig,
                offset_orig = 0;
                for ( i = 0, i < pic_old->dim; i++ )
                {
                    offset_orig += length_vect [i] *
                      (( axis_vect [i] == 0 ) ?
                       index_vect [i] :
                      (axis_vect * index_vect[i]));
                }
```

The macro `FORLOOP` is defined as following

```
#define FORLOOP( type, pic_new, index, offset_orig,     \
              offset_func )                             \
{                                                       \
  ipUInt4_t   i;                                        \
  ipUInt4_t   offset_Refl;                              \
  ipInt4_t    n[_ipPicNDIM];                            \
                                                        \
  for ( i = 0; i < pic_old->dim; i++ )                  \
    n[i] = pic_old->n[i];                               \
  for ( i = pic_oldpic_oldpic->dim; i < _ipPicNDIM; i++
)                                                       \
   n[i] = 1;                                            \
                                                        \
  offset_Refl = 0;                                      \
```

```
  for ( index[7] = 0; index[7] < n[7], index[7]++ )      \
    for ( index[6] = 0; index[6]  < n[6]; index[6]++ )   \
      for ( index[5] = 0; index[5] <  n[5];index[5]++)   \
        for ( index[4] = 0; index[4] < n[4];index[4]++) \
          for ( index[3] = 0; index[3] < n[3];           \
                                index[3]++ )             \
            for ( index[2] = 0; index[2] < n[2];         \
                                  index[2]++ )           \
              for ( index[1] = 0; index[1] < n[1];       \
                                    index[1]++ )         \
                for ( index[0] = 0; index[0] < n[0];     \
                                      index[0]++ )       \
                {                                        \
                  offset_func;                           \
                  (( type * ) pic_new->data )            \
                                       [offset_Refl] =   \
                     (( type * ) pic->data )             \
                                       [offset_orig];    \
                  offset_orig++;                         \
                }                                        \
}
```

Where `offset_func` contains the algorithm to calculate the offset of the pixel which has to be changed with the current one.

Another possibility to realize the nested for loops is to use a switch statement. Depending on the number of dimensions the program branches to the corresponding loop and just executes the loops with an index less then the number of dimensions. This macro isn't listed completely in this document. Just the general structure of this switch statement will be shown. Both macros (`ipFuncFORALL` and FORLOOP) have the same interface.

```
#define ipFuncFORALL ( type, pic_old,pic_new, index,
                       offset_orig, offset_func)
...
switch ( pic->dim )
{
  case 8:
    for ( index[8] = 0; ...
  case 7:
      for ( index[7] = 0; ...
  case 6:
        for ( index[6] = 0; ...
  ...
  case 0:
          for ( index[0] = 0; ...
          {
            /* transformation of image */
```

```
                }
...
```

A version of this macro is in file ipFunc.h in directory `/usr/local/include/ip`

4 Literature

[Engelmann 1994] Engelmann U, Schröter A. *PIC image file standard*. Technical Report TR 48 MBI/DKFZ 1994.

[Schröter 1994] Schröter *A. PIC C Library 3.0*. Technical Report MBI/DKFZ 1994.

[Schröter 1994] Schröter A. *MBI C Coding Standards.* Technical Report TR 56 MBI/DKFZ 1994.

A Program Listings

In this chapter all the programs described in the previous chapters are listed. They are stored in the directory /usr/users/antje/fertig/. The name of the file is added in brackets in the name of the chapters. As well for the inversion as for the reflection there is a main program and an include file, which could be used for all versions of each program. In this dirctory there is also a makefile which could be used to compile all programs.

A.1 Inversion of an image

A.1.1 Main program (main_inv.c)

```
#include "invert.h"

void main ( int argc, char *argv[] )
{

/* variables                                                          */

ipPicDescriptor *pic, *pic_new;
char          pic_name[NAME_MAX];     /* file name of original picture */
char          pic_name_inv[NAME_MAX]; /* file name of inv. picture     */
int           pos;                    /* position in file_name where   */
                                      /* ".pic" begins                 */

/* read a picture                                                     */

if ( argc == 1 )
  {
     printf ( "picture name   : ");
     scanf  ( "%s", pic_name);
   }
else  if (argc == 2 )
  {
    strcpy ( pic_name, argv[1] );
  }

pic = ipPicGet( pic_name, NULL );
if ( pic != NULL )
   {
     /* invert  the picture in function picInv                        */
```

```
      pic_new = picInv ( pic );

      /* show original picture                                          */

      ipPicPut ( pic_name, pic );
      if ( fork() == 0 )
        execl ( "/usr/local/bin/movie", "movie", pic_name, NULL );

      /* show reflected picture                                         */

      pos = strstr ( pic_name, ".pic" ) - pic_name;
      strncpy ( pic_name_inv, pic_name, pos );
      strcat ( pic_name_inv, ".inv.pic" );
      ipPicPut ( pic_name_inv, pic_new );
      if ( fork() == 0 )
        execl ( "/usr/local/bin/movie", "movie", pic_name_inv, NULL );
    }
  else printf ( "Error in reading an image \n");
  }
```

A.1.2 Include file (invert.h)

```
#ifndef _invert_h
#define _invert_h

/* include files                                                       */

#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <ipPic/ipPic.h>
#include <ip/ipFunc.h>

/* function prototypes                                                 */

ipPicDescriptor *picInv ( ipPicDescriptor *pic_old );

#endif /* _invert_h */
/* DON'T ADD ANYTHING AFTER THIS #endif                                */
```

A.1.3 Function invert.c (without using macros)

```
/* ------------------------------------------------------------------*/
/*
**  function picInv : inverts the greyvalues of the picture (pic_old)
**                 and returns the inverted picture (pic_new)
*/
/* ------------------------------------------------------------------*/

#include"invert.h"
ipPicDescriptor *picInv ( ipPicDescriptor *pic_old )
{
  ipPicDescriptor *pic_new;  /* inverted picture                     */
  ipUInt4_t      i, j;       /* loopindex                            */
```

```
/* create a new picture, copy the header, allocate memory            */

pic_new = ipPicCopyHeader ( pic_old, NULL );
pic_new->data = malloc ( _ipPicSize ( pic_new ) );

/* type of the pictureelements and changing of data                  */

switch ( ipPicDR ( pic_old->type, pic_old->bpe ) )
  {

    /* Integer (1 Byte)                                              */

    case ipPicDR( ipPicInt, 8 ) :

      ipInt1_t help;

      /* max. und min. greyvalue                                     */

      ipInt1_t  max = (( ipInt1_t * )pic_old->data ) [0];
      ipInt1_t  min = (( ipInt1_t * )pic_old->data ) [0];

      for ( i = 1; i < _ipPicElements (pic_old); i++ )
        {
          help = (( ipInt1_t * )pic_old->data ) [i];
          max  = ( help > max ) ? help : max;
          min  = ( help < min ) ? help : min;
        }

      /* invert greyvalues                                           */

      for ( i = 0; i < _ipPicElements (pic_old); i++ )
        {
          (( ipInt1_t * )pic_new->data ) [i] =
               max + min - (( ipInt1_t * )pic_old->data ) [i];
        }
      break;

    /* Integer (2 Byte)                                              */

    case ipPicDR( ipPicInt, 16 ) :

      ipInt2_t help;

      /* max. und min. greyvalue                                     */

      ipInt2_t  max = (( ipInt2_t * )pic_old->data ) [0];
      ipInt2_t  min = (( ipInt2_t * )pic_old->data ) [0];

      for ( i = 1; i < _ipPicElements (pic_old); i++ )
        {
          help = (( ipInt1_t * )pic_old->data ) [i];
          max  = ( help > max ) ? help : max;
          min  = ( help < min ) ? help : min;
        }

      /* invert greyvalues                                           */

      for ( i = 0; i < _ipPicElements (pic_old); i++ )
        {
```

```
        (( ipInt2_t * )pic_new->data ) [i] =
             max + min - (( ipInt2_t * )pic_old->data ) [i];
      }
    break;

  /* Integer (4 Byte)                                              */

  case ipPicDR( ipPicInt, 32 ) :

    ipInt4_t help;

    /* max. und min. greyvalue                                     */

    ipInt4_t  max = (( ipInt4_t * )pic_old->data ) [0];
    ipInt4_t  min = (( ipInt4_t * )pic_old->data ) [0];

    for ( i = 1; i < _ipPicElements (pic_old); i++ )
      {
        help = (( ipInt4_t * )pic_old->data ) [i];
        max  = ( help > max ) ? help : max;
        min  = ( help < min ) ? help : min;
      }

    /* invert greyvalues                                           */

    for ( i = 0; i < _ipPicElements (pic_old); i++ )
      {
        (( ipInt4_t * )pic_new->data ) [i] =
             max + min - (( ipInt4_t * )pic_old->data ) [i];
      }

    break;

  /* Unsigned  Integer (1 Byte)                                    */

  case ipPicDR( ipPicUInt, 8 ) :

    ipUInt1_t help;

    /* max. und min. greyvalue                                     */

    ipUInt1_t  max = (( ipUInt1_t * )pic_old->data ) [0];
    ipUInt1_t  min = (( ipUInt1_t * )pic_old->data ) [0];

    for ( i = 1; i < _ipPicElements (pic_old); i++ )
      {
        help = (( ipUInt1_t * )pic_old->data ) [i];
        max  = ( help > max ) ? help : max;
        min  = ( help < min ) ? help : min;
      }

    /* invert greyvalues                                           */

    for ( i = 0; i < _ipPicElements (pic_old); i++ )
      {
        (( ipUInt1_t * )pic_new->data ) [i] =
             max + min - (( ipUInt1_t * )pic_old->data ) [i];
      }

    break;
```

```
/* Unsigned  Integer (2 Byte)                                        */

case ipPicDR( ipPicUInt, 16 ) :

  ipUInt2_t help;

  /* max. und min. greyvalue                                         */

  ipUInt2_t  max = (( ipUInt2_t * )pic_old->data ) [0];
  ipUInt2_t  min = (( ipUInt2_t * )pic_old->data ) [0];

  for ( i = 1; i < _ipPicElements (pic_old); i++ )
    {
      help = (( ipUInt2_t * )pic_old->data ) [i];
      max  = ( help > max ) ? help : max;
      min  = ( help < min ) ? help : min;
    }

  /* invert greyvalues                                               */

  for ( i = 0; i < _ipPicElements (pic_old); i++ )
    {
      (( ipUInt2_t * )pic_new->data ) [i] =
           max + min - (( ipUInt2_t * )pic_old->data ) [i];
    }

  break;

/* Unsigned  Integer (4 Byte)                                        */

case ipPicDR( ipPicUInt, 32 ) :

  ipUInt4_t help;

  /* max. und min. greyvalue                                         */

  ipUInt1_t  max = (( ipUInt4_t * )pic_old->data ) [0];
  ipUInt4_t  min = (( ipUInt4_t * )pic_old->data ) [0];

  for ( i = 1; i < _ipPicElements (pic_old); i++ )
    {
      help = (( ipUInt4_t * )pic_old->data ) [i];
      max  = ( help > max ) ? help : max;
      min  = ( help < min ) ? help : min;
    }

  /* invert greyvalues                                               */

  for ( i = 0; i < _ipPicElements (pic_old); i++ )
    {
      (( ipUInt4_t * )pic_new->data ) [i] =
           max + min - (( ipUInt4_t * )pic_old->data ) [i];
    }

  break;

/* Float (4 Byte)                                                    */

case ipPicDR( ipPicFloat, 32 ) :
```

```
            ipFloat4_t help;

            /* max. und min. greyvalue                                    */

            ipFloat4_t  max = (( ipFloat4_t * )pic_old->data ) [0];
            ipFloat4_t  min = (( ipFloat4_t * )pic_old->data ) [0];

            for ( i = 1; i < _ipPicElements (pic_old); i++ )
              {
                help = (( ipFloat4_t * )pic_old->data ) [i];
                max  = ( help > max ) ? help : max;
                min  = ( help < min ) ? help : min;
              }

            /* invert greyvalues                                          */

            for ( i = 0; i < _ipPicElements (pic_old); i++ )
              {
                (( ipFloat4_t * )pic_new->data ) [i] =
                     max + min - (( ipFloat4_t * )pic_old->data ) [i];
              }

            break;

          /* Float (8 byte)                                               */

          case ipPicDR( ipPicFloat, 64 ) :

            ipUFloat8_t help;

            /* max. und min. greyvalue                                    */

            ipFloat8_t  max = (( ipFloat8_t * )pic_old->data ) [0];
            ipFloat8_t  min = (( ipFloat8_t * )pic_old->data ) [0];

            for ( i = 1; i < _ipPicElements (pic_old); i++ )
              {
                help = (( ipFloat8_t * )pic_old->data ) [i];
                max  = ( help > max ) ? help : max;
                min  = ( help < min ) ? help : min;
              }

            /* invert greyvalues                                          */

            for ( i = 0; i < _ipPicElements (pic_old); i++ )
              {
                (( ipFloat8_t * )pic_new->data ) [i] =
                     max + min - (( ipFloat8_t * )pic_old->data ) [i];
              }

            break;

          default :
            printf ( " Data type doesn"t exist \n " );

        }
      }
```

A.1.4 Function invert_m.c (using macros)

```
                                                                        */
/* include-Files                                                        */

#include "invert.h"
                                                                        */
/* definition of invert-macro                                           */
                                                                        */
#define INV( type, pic )                                                 \
{                                                                        \
  ipUInt4_t  i;                                                          \
  type     help;                                                         \
                                                                         \
  type  max = ( ( type * ) pic->data ) [0];                              \
  type  min = ( ( type * ) pic->data ) [0];                              \
                                                                         \
  for ( i = 1; i < _ipPicElements ( pic ); i++ )                         \
   {                                                                     \
     help = ( ( type * ) pic->data ) [i];                                \
     max  = ( help > max ) ? help : max;                                 \
     min  = ( help < min ) ? help : min;                                 \
   }                                                                     \
                                                                         \
  for ( i = 0; i < _ipPicElements ( pic ); i++ )                         \
   {                                                                     \
     (( type * ) pic_new->data ) [i] =                                   \
         max + min - (( type * ) pic->data ) [i];                        \
   }                                                                     \
}                                                                        \



/* ---------------------------------------------------------------  */
/*
**  function picInv : inverts the greyvalues of the picture (pic_old)
**                and returns the inverted picture (pic_new)
*/
/* ---------------------------------------------------------------  */

ipPicDescriptor *picInv ( ipPicDescriptor *pic_old )
{

  ipPicDescriptor *pic_new;  /* inverted picture                    */
  ipUInt4_t      i, j;     /* loopindex                             */
  ipUInt1_t      laenge;    /* length of a pictureelement           */


  /* create a new picture, copy the header, allocate memory         */

  pic_new = ipPicCopyHeader ( pic_old, 0 );
  pic_new->data = malloc ( _ipPicSize ( pic_new ) );

  /* macro to invert the picture (for all data types)               */
```

```
  ipPicFORALL ( INV, pic_old );

  return pic_new;
}
```

A.2 Reflection of an image

A.2.1 Main program (main_refl.c)

```
/* include files                                                    */

#include "reflect.h"

void main ( int argc, char *argv[] )
{

  /* variables                                                      */

  ipPicDescriptor *pic, *pic_new;
  char          pic_name[NAME_MAX]; /* file name of original picture   */
  char          pic_name_refl[NAME_MAX];/* file name of refl picture   */
  ipInt4_t       achse;            /* reflection-axis                 */
  ipInt4_t       pos;              /* position in file_name where     */
                                   /* ".pic" begins                   */

  /* read a picture in                                              */

  if ( argc == 1 )
   {
      printf ( "image name : " );
      scanf  ( "%s", pic_name);
      printf ( "reflection axis : " );
      scanf  ( "%d", &achse );
   }
  else  if (argc == 2 )
   {
      strcpy ( pic_name, argv[1] );
      printf ( "reflection axis : " );
      scanf  ( "%d", &achse );
   }
  else  if (argc == 3 )
   {
      strcpy ( pic_name, argv[1] );
      sscanf( argv[2], "%d", &achse );
   }

  pic = ipPicGet( pic_name, NULL );
  if ( pic != NULL )
   {

      /* reflect the picture in function picRefl                    */

       pic_new = picRefl ( pic, achse );

      /* show original picture                                      */
```

```
     ipPicPut ( pic_name, pic );
     if ( fork() == 0 )
        execl ( "/usr/local/bin/movie", "movie", pic_name, NULL );

     /* show reflected picture                                          */

     pos = strstr ( pic_name, ".pic" ) - pic_name;
     strncpy ( pic_name_refl, pic_name, pos );
     strcat ( pic_name_refl, ".Refl.pic" );

     ipPicPut ( pic_name_refl, pic_new );
     if ( fork() == 0 )
        execl ( "/usr/local/bin/movie", "movie", pic_name_refl, NULL );
   }
  else printf ( "Error in reading an image    \n" );
}
```

A.2.2 Include file (reflect.h)

```
#ifndef _reflect_h
#define _reflect_h

/* include files                                                        */

#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <ipPic/ipPic.h>
#include <ip/ipFunc.h>

/* defining constants                                                   */

#define DIMMAX 4

/* function prototypes                                                  */

ipPicDescriptor *picRefl ( ipPicDescriptor *pic_old,
                           ipInt4_t       axis );

 #endif /* _invert_h */
/* DON'T ADD ANYTHING AFTER THIS #endif                                 */
```

A.2.3 Function reflect2.c (generalized algorithm)

```
/* include files                                                        */

#include "reflect.h"


/* ------------------------------------------------------------------  */
/*
**  function picRefl :
*/
/* ------------------------------------------------------------------  */

ipPicDescriptor *picRefl ( ipPicDescriptor *pic_old, ipInt4_t achse )
```

```
{
                                                                        */
  ipPicDescriptor *pic_new;            /* inverted picture            */
  ipUInt4_t      index_vect[_ipPicNDIM]; /* loopindex-vector           */
  ipUInt4_t      length_vect[_ipPicNDIM];
  ipUInt4_t      axis_vect[_ipPicNDIM];
  ipInt4_t       n[_ipPicNDIM];
  ipUInt4_t      i, j;
  ipUInt4_t      offset_orig;
  ipUInt4_t      offset_refl;

  /* initialisation of vectors                                         */

  for ( i = 0; i < pic_old->dim; i++ )
    n[i] = pic_old->n[i];

  for ( i = pic_old->dim; i < _ipPicNDIM; i++ )
    n[i] = 1;

  for ( i = 0; i < _ipPicNDIM; i++ )
    {
      index_vect[i] = 0;
      axis_vect[i] = 0;
    }

  if ( achse == 1 )
    achse = 2;
  else if ( achse == 2 )
    achse = 1;
  axis_vect[achse - 1] = n[achse - 1] - 1;

  length_vect[0] = 1;
  for ( i = 1; i < pic_old->dim; i++ )
    length_vect[i] = length_vect[i-1] * n[i];

  /* create a new picture, copy the header, allocate memory            */

  pic_new =  ipPicCopyHeader ( pic_old, NULL );
  pic_new->data = malloc ( _ipPicSize ( pic_new ) );

  /* type of the pictureelements and changing of data                  */

  switch ( ipPicDR( pic_old->type, pic_old->bpe ) )
    {

    /* Integer (1 Byte)                                                */

      case ipPicDR( ipPicInt, 8 ) :
        offset_refl = 0;
        for ( index_vect[7] = 0  index_vect[7] < n[7];
                                 index_vect[7]++ )
          for ( index_vect[6] = 0;  index_vect[6] < n[6];
                                    index_vect[6]++ )
            for ( index_vect[5] = 0; index_vect[5] < n[5];
                                     index_vect[5]++ )
              for ( index_vect[4] = 0; index_vect[4] < n[4];
                                       index_vect[4]++ )
                for ( index_vect[3] = 0; index_vect[3] < n[3];
                                         index_vect[3]++ )
                  for ( index_vect[2] = 0; index_vect[2] < n[2];
```

```
                                        index_vect[2]++ )
                    for ( index_vect[1] = 0; index_vect[1] < n[1];
                                          index_vect[1]++ )
                      for ( index_vect[0] = 0; index_vect[0] < n[0];
                                             index_vect[0]++ )
                        {
                          offset_orig = 0;
                          for ( i = 0; i < pic_old->dim; i++ )
                            {
                              offset_orig = offset_orig +
                                 length_vect[i] *
                                (( axis_vect[i] == 0 ) ?
                                index_vect[i] :
                                ( axis_vect[i] - index_vect[i]));
                            }
                          (( ipInt1_t * ) pic_new->data )
                                                  [offset_refl] =
                            (( ipInt1_t * ) pic_old->data )
                                                  [offset_orig];
                          offset_refl++;
                        }
  break;

/* Integer (2 Byte)                                                 */

case ipPicDR( ipPicInt, 16 ) :
  offset_refl = 0;
  for ( index_vect[7] = 0  index_vect[7] < n[7];
                           index_vect[7]++ )
    for ( index_vect[6] = 0;  index_vect[6] < n[6];
                            index_vect[6]++ )
      for ( index_vect[5] = 0; index_vect[5] < n[5];
                              index_vect[5]++ )
        for ( index_vect[4] = 0; index_vect[4] < n[4];
                                index_vect[4]++ )
          for ( index_vect[3] = 0; index_vect[3] < n[3];
                                  index_vect[3]++ )
            for ( index_vect[2] = 0; index_vect[2] < n[2];
                                  index_vect[2]++ )
              for ( index_vect[1] = 0; index_vect[1] < n[1];
                                    index_vect[1]++ )
                for ( index_vect[0] = 0; index_vect[0] < n[0];
                                       index_vect[0]++ )
                  {
                    offset_orig = 0;
                    for ( i = 0; i < pic_old->dim; i++ )
                      {
                        offset_orig = offset_orig +
                           length_vect[i] *
                          (( axis_vect[i] == 0 ) ?
                          index_vect[i] :
                          ( axis_vect[i] - index_vect[i]));
                      }
                    (( ipInt2_t * ) pic_new->data )
                                                  [offset_refl] =
                      (( ipInt2_t * ) pic_old->data )
                                                  [offset_orig];
                    offset_refl++;
                  }
  break;
```

```
/* Integer (4 Byte)                                                  */

case ipPicDR( ipPicInt, 32 ) :
  offset_refl = 0;
  for ( index_vect[7] = 0  index_vect[7] < n[7];
                           index_vect[7]++ )
    for ( index_vect[6] = 0;  index_vect[6] < n[6];
                              index_vect[6]++ )
      for ( index_vect[5] = 0; index_vect[5] < n[5];
                                 index_vect[5]++ )
        for ( index_vect[4] = 0; index_vect[4] < n[4];
                                   index_vect[4]++ )
          for ( index_vect[3] = 0; index_vect[3] < n[3];
                                     index_vect[3]++ )
            for ( index_vect[2] = 0; index_vect[2] < n[2];
                                     index_vect[2]++ )
              for ( index_vect[1] = 0; index_vect[1] < n[1];
                                       index_vect[1]++ )
                for ( index_vect[0] = 0; index_vect[0] < n[0];
                                         index_vect[0]++ )
                  {
                    offset_orig = 0;
                    for ( i = 0; i < pic_old->dim; i++ )
                      {
                        offset_orig = offset_orig +
                           length_vect[i] *
                          (( axis_vect[i] == 0 ) ?
                          index_vect[i] :
                          ( axis_vect[i] - index_vect[i]));
                      }
                    (( ipInt4_t * ) pic_new->data )
                                           [offset_refl] =
                      (( ipInt4_t * ) pic_old->data )
                                           [offset_orig];
                    offset_refl++;
                  }
  break;

/* Unsigned Integer (1 Byte)                                         */

case ipPicDR( ipPicUInt, 8 ) :
  offset_refl = 0;
  for ( index_vect[7] = 0  index_vect[7] < n[7];
                           index_vect[7]++ )
    for ( index_vect[6] = 0;  index_vect[6] < n[6];
                              index_vect[6]++ )
      for ( index_vect[5] = 0; index_vect[5] < n[5];
                                 index_vect[5]++ )
        for ( index_vect[4] = 0; index_vect[4] < n[4];
                                   index_vect[4]++ )
          for ( index_vect[3] = 0; index_vect[3] < n[3];
                                     index_vect[3]++ )
            for ( index_vect[2] = 0; index_vect[2] < n[2];
                                     index_vect[2]++ )
              for ( index_vect[1] = 0; index_vect[1] < n[1];
                                       index_vect[1]++ )
                for ( index_vect[0] = 0; index_vect[0] < n[0];
                                         index_vect[0]++ )
                  {
```

```
                              offset_orig = 0;
                              for ( i = 0; i < pic_old->dim; i++ )
                                 {
                                    offset_orig = offset_orig +
                                       length_vect[i] *
                                      (( axis_vect[i] == 0 ) ?
                                      index_vect[i] :
                                      ( axis_vect[i] - index_vect[i]));
                                 }
                              (( ipUInt1_t * ) pic_new->data )
                                                      [offset_refl] =
                                 (( ipUInt1_t * ) pic_old->data )
                                                      [offset_orig];
                              offset_refl++;
                            }
      break;

    /*Unsigned  Integer (2 Byte)

    case ipPicDR( ipPicUInt, 16 ) :
      offset_refl = 0;
      for ( index_vect[7] = 0  index_vect[7] < n[7];
                               index_vect[7]++ )
        for ( index_vect[6] = 0;  index_vect[6] < n[6];
                                  index_vect[6]++ )
          for ( index_vect[5] = 0; index_vect[5] < n[5];
                                    index_vect[5]++ )
            for ( index_vect[4] = 0; index_vect[4] < n[4];
                                      index_vect[4]++ )
              for ( index_vect[3] = 0; index_vect[3] < n[3];
                                        index_vect[3]++ )
                for ( index_vect[2] = 0; index_vect[2] < n[2];
                                        index_vect[2]++ )
                  for ( index_vect[1] = 0; index_vect[1] < n[1];
                                           index_vect[1]++ )
                    for ( index_vect[0] = 0; index_vect[0] < n[0];
                                             index_vect[0]++ )
                      {
                        offset_orig = 0;
                        for ( i = 0; i < pic_old->dim; i++ )
                          {
                            offset_orig = offset_orig +
                               length_vect[i] *
                              (( axis_vect[i] == 0 ) ?
                              index_vect[i] :
                              ( axis_vect[i] - index_vect[i]));
                          }
                        (( ipUInt2_t * ) pic_new->data )
                                                      [offset_refl] =
                          (( ipUInt2_t * ) pic_old->data )
                                                      [offset_orig];
                        offset_refl++;
                      }
      break;

  /* Unsigned Integer (4 Byte)                                        */

    case ipPicDR( ipPicUInt, 32 ) :
      offset_refl = 0;
      for ( index_vect[7] = 0  index_vect[7] < n[7];
```

```
                              index_vect[7]++ )
        for ( index_vect[6] = 0;  index_vect[6] < n[6];
                                 index_vect[6]++ )
          for ( index_vect[5] = 0; index_vect[5] < n[5];
                                   index_vect[5]++ )
            for ( index_vect[4] = 0; index_vect[4] < n[4];
                                     index_vect[4]++ )
              for ( index_vect[3] = 0; index_vect[3] < n[3];
                                       index_vect[3]++ )
                for ( index_vect[2] = 0; index_vect[2] < n[2];
                                         index_vect[2]++ )
                  for ( index_vect[1] = 0; index_vect[1] < n[1];
                                           index_vect[1]++ )
                    for ( index_vect[0] = 0; index_vect[0] < n[0];
                                             index_vect[0]++ )
                      {
                        offset_orig = 0;
                        for ( i = 0; i < pic_old->dim; i++ )
                          {
                            offset_orig = offset_orig +
                               length_vect[i] *
                              (( axis_vect[i] == 0 ) ?
                              index_vect[i] :
                              ( axis_vect[i] - index_vect[i]));
                          }
                        (( ipUInt4_t * ) pic_new->data )
                                                 [offset_refl] =
                          (( ipUInt4_t * ) pic_old->data )
                                                 [offset_orig];
                        offset_refl++;
                      }
  break;

/* Float (4 Byte)                                                 */

case ipPicDR( ipPicFloat, 32 ) :
  offset_refl = 0;
  for ( index_vect[7] = 0  index_vect[7] < n[7];
                           index_vect[7]++ )
    for ( index_vect[6] = 0;  index_vect[6] < n[6];
                              index_vect[6]++ )
      for ( index_vect[5] = 0; index_vect[5] < n[5];
                               index_vect[5]++ )
        for ( index_vect[4] = 0; index_vect[4] < n[4];
                                 index_vect[4]++ )
          for ( index_vect[3] = 0; index_vect[3] < n[3];
                                   index_vect[3]++ )
            for ( index_vect[2] = 0; index_vect[2] < n[2];
                                     index_vect[2]++ )
              for ( index_vect[1] = 0; index_vect[1] < n[1];
                                       index_vect[1]++ )
                for ( index_vect[0] = 0; index_vect[0] < n[0];
                                         index_vect[0]++ )
                  {
                    offset_orig = 0;
                    for ( i = 0; i < pic_old->dim; i++ )
                      {
                        offset_orig = offset_orig +
                           length_vect[i] *
                          (( axis_vect[i] == 0 ) ?
```

```
                                        index_vect[i] :
                                        ( axis_vect[i] - index_vect[i]));
                                    }
                                  (( ipFloat4_t * ) pic_new->data )
                                                          [offset_refl] =
                                    (( ipFloat4_t * ) pic_old->data )
                                                          [offset_orig];
                                  offset_refl++;
                                }
          break;

        /* Float (8 Byte)                                                 */

        case ipPicDR( ipPicFloat, 64 ) :
          offset_refl = 0;
          for ( index_vect[7] = 0  index_vect[7] < n[7];
                                   index_vect[7]++ )
            for ( index_vect[6] = 0;  index_vect[6] < n[6];
                                      index_vect[6]++ )
              for ( index_vect[5] = 0; index_vect[5] < n[5];
                                       index_vect[5]++ )
                for ( index_vect[4] = 0; index_vect[4] < n[4];
                                         index_vect[4]++ )
                  for ( index_vect[3] = 0; index_vect[3] < n[3];
                                           index_vect[3]++ )
                    for ( index_vect[2] = 0; index_vect[2] < n[2];
                                             index_vect[2]++ )
                      for ( index_vect[1] = 0; index_vect[1] < n[1];
                                               index_vect[1]++ )
                        for ( index_vect[0] = 0; index_vect[0] < n[0];
                                                 index_vect[0]++ )
                          {
                            offset_orig = 0;
                            for ( i = 0; i < pic_old->dim; i++ )
                              {
                                offset_orig = offset_orig +
                                   length_vect[i] *
                                  (( axis_vect[i] == 0 ) ?
                                  index_vect[i] :
                                  ( axis_vect[i] - index_vect[i]));
                              }
                            (( ipFloat8_t * ) pic_new->data )
                                                          [offset_refl] =
                              (( ipFloat_t * ) pic_old->data )
                                                          [offset_orig];
                            offset_refl++;
                          }
          break;

        default :
          printf ( "datatype doesn’t exist \n" );
      }
    return pic_new;
  }
```

A.2.4 Function reflect_m2.c (generalized, minimized version)

```
/* include Files

#include "reflect.h"                                                    */

/* definition of reflect-macro                                          */

#define ForLoop( type, pic, pic_new, index, offset_orig, offset_func )   \
{                                                                        \
  ipUInt4_t   i;                                                         \
  ipUInt4_t   offset_refl;                                               \
  ipInt4_t    n[_ipPicNDIM];                                             \
                                                                         \
  for ( i = 0; i < pic_old->dim; i++ )                                   \
    n[i] = pic_old->n[i];                                                \
                                                                         \
  for ( i = pic_old->dim; i < _ipPicNDIM; i++ )                          \
    n[i] = 1;                                                            \
                                                                         \
  offset_refl = 0;                                                       \
  for ( index[7] = 0; index[7] < n[7]; index[7]++ )                      \
                                                                         \
    for ( index[6] = 0; index[6] < n[6]; index[6]++ )                    \
                                                                         \
      for ( index[5] = 0; index[5] < n[5]; index[5]++ )                  \
                                                                         \
        for ( index[4] = 0; index[4] < n[4]; index[4]++ )                \
                                                                         \
          for ( index[3] = 0; index[3] < n[3]; index[3]++ )              \
                                                                         \
            for ( index[2] = 0; index[2] < n[2]; index[2]++ )            \
                                                                         \
              for ( index[1] = 0; index[1] < n[1]; index[1]++ )          \
                                                                         \
                for ( index[0] = 0; index[0] < n[0]; index[0]++ )        \
                {                                                        \
                  offset_func;                                           \
                  (( type * ) pic_new->data )  [offset_refl] =           \
                     (( type * ) pic->data ) [offset_orig];              \
                  offset_refl++;                                         \
                }                                                        \
}                                                                        \



/* ------------------------------------------------------------------  */
/*
**  function picRefl : reflects a picture (pic_old) along one axis
**                 (axis)
**                 axis: 1 => x-axis
**                       2 => y-axis
**                       3 => z-axis
**                       4 => t-axis
*/
/* ------------------------------------------------------------------  */
```

```
ipPicDescriptor *picRefl ( ipPicDescriptor *pic_old, int axis )
{

  ipPicDescriptor *pic_new;              /* inverted picture          */
  ipUInt4_t      index_vect[_ipPicNDIM]; /* loopindex-vector          */
  ipUInt4_t      length_vect[_ipPicNDIM];
  ipUInt4_t      axis_vect[_ipPicNDIM];
  ipInt4_t       n[_ipPicNDIM];          /* number of pixels in each  */
                                         /* dimension                 */
  ipUInt4_t      i, j;                   /* loop index                */
  ipUInt4_t      offset_orig;

  /* initialisation of vectors                                       */

  for ( i = 0; i < pic_old->dim; i++ )
     n[i] = pic_old->n[i];

  for ( i = pic_old->dim; i < _ipPicNDIM; i++ )
    n[i] = 1;


  for ( i = 0; i < _ipPicNDIM; i++ )
   {
     index_vect[i] = 0;
     axis_vect[i] = 0;
   }

  if ( axis == 1 )
     axis = 2;
  else if ( axis == 2 )
     axis = 1;
  axis_vect[axis - 1] = n[axis - 1] - 1;

  length_vect[0]  = 1;
  for ( i = 1; i < pic_old->dim; i++ )
     length_vect[i] = n[i] * length_vect[i-1];

  /* create a new picture, copy the header, allocate memory          */

  pic_new = ipPicCopyHeader ( pic_old, 0 );
  pic_new->data = malloc ( _ipPicSize ( pic_new ) );

  ipPicFORALL_4  ( ForLoop, pic_old, pic_new, index_vect, offset_orig,
                   offset_orig = 0;
                   for ( i = 0; i < pic_old->dim; i++ )
                      {
                         offset_orig = offset_orig + length_vect [i] *
                             (( axis_vect [i] == 0 ) ?
                                index_vect [i] :
                                ( axis_vect [i] - index_vect[i] ));
                      }
                )

  return pic_new;
}
```

A.3 Makefile

```
#
# compiler (letzter gilt)
#
# C: ULTIRX(borneo, bali), Alpha(sumatra)
CC = c89

# C: Linux(harris), SGI(celebes, marvin), SunOS(zaphod)
CC = cc

# libraries
# pic library
PIC_LIB  = -lipPic

# include file path
INCS    = -I/usr/local/include

# all libararies
LIB     = $(PIC_LIB) -lm

#
# Compiler Options
#
DEBUG = -g
OPTIM = -O
CFLAGS =  $(OPTIM) $(INCS)
CFLAGS =  $(DEBUG) $(INCS)

LDFLAGS = $(OPTIM)
LDFLAGS = $(DEBUG)

# Compilieren der .c files --> .o files
#
.SUFFIXES: .o .c
.c.o:
   $(CC) $(CFLAGS) -c $<

all : invert invert_m reflect reflect2 reflect_m2 reflect_m3


invert: main_inv.o invert.o
   $(CC) $(LDFLAGS) -o $@ main_inv.o invert.o $(LIB)

invert_m: main_inv.o invert_m.o
   $(CC) $(LDFLAGS) -o $@ main_inv.o invert_m.o $(LIB)

reflect: main_refl.o reflect.o
   $(CC) $(LDFLAGS) -o $@ main_refl.o reflect.o $(LIB)

reflect2: main_refl.o reflect2.o
   $(CC) $(LDFLAGS) -o $@ main_refl.o reflect2.o $(LIB)

reflect_m2: main_refl.o reflect_m2.o
   $(CC) $(LDFLAGS) -o $@ main_refl.o reflect_m2.o $(LIB)

reflect_m3: main_refl.o reflect_m3.o
   $(CC) $(LDFLAGS) -o $@ main_refl.o reflect_m3.o $(LIB)

main_inv.o    : invert.h
```

```
invert.o       : invert.h
invert_m.o     : invert.h

main_refl.o    : reflect.h
reflect.o      : reflect.h
reflect2.o     : reflect.h
reflect_m2.o   : reflect.h
reflect_m3.o   : reflect.h
```