The BIFF Image Concept, File Format and Routine Library

Tor Lønnestad

2. edition, February 1990

Contents

Chapter 1

Introduction

Image processing is a field involving large amounts of data, and numerous algorithms for operating on these data. In order to get as far as possible with limited human and machine resources, the exchange of data as well as software between different people, different computers, and different institutions, is essential. To simplify this communication, standards are necessary. This paper describes a subroutine library to handle the image standard used at the image processing laboratory at the Department of Informatics, University of Oslo. The paper also gives a brief description of the standard itself. This edition has only minor changes from the first edition in October 1988.

There are several ways to represent images in a much more compact way than raster format. Surveys are found in [1] and [2]. The main advantage of the raster format is simplicity, most algorithms are defined to work on raster images and there is no need for coding – decoding. This report will only deal with raster images.

1.1 Existing file formats

The need for standards have long since been recognized, and several such standards have been proposed and are in use. However, most of them belong to one of the following groups:

- The file format is defined as a transport format, not a storage format. The format is designed for simple transport to and from magnetic tape [3], or simple transfer by telecommunication [4], but in either case it is usually not suited for reading into a program by random access.

- The file format is a part of a large, complete commercial software (and hardware) system. The file format is considered internal to the system, and documentation is poor or missing [5], [6].

- The file format is highly application specific, either because of specific information fields, or because of size restrictions [7].

The few formats left when the previous groups are excluded, are generally marked by age. Because of memory limitations and lack of virtual memory, it was considered impossible to handle a whole band at a time. Thus, lots of work has been put down in the decomposition of a band into blocks of acceptable size [8], [9], [10], [11]. In most of these systems the user have to face this complexity, and call the appropriate read- and write-routines at block boundaries. Alternatively, routines could transform band coordinates to block coordinates and automatically fetch blocks from file when needed, at the cost of run-time performance. Neither of these solutions are good, and fortunately, as computers are equipped with more and more memory, the need for blocking decreases.

A property common to most existing image file formats, is the total absence of a conceptual image definition. Libraries for handling the formats are generally written in FORTRAN, if implemented at all. Suitable datastructures for representing images inside programs are left to the user to define.

File formats from any of these groups are not very suited for general applications, and there seems to be a need for a format with the following properties:

- Designed for storage and retrieval of raster images.

- Large freedom in sizes, pixel types etc. and thus application independent.

- Operating system and programming language independent image definition.

- Operating system and programming language independent file format.

- Routine library with suitable data types for C under UNIX.

A more detailed design-recommendation can be found in [12].

1.2 Organization of the report

The word BIFF is used in three ways throughout this report.

- First of all it stands for Blab Image File Format, where "Blab" is a Norwegian short form of image processing laboratory.
- BIFF is also an image concept, meaning that it is a way of thinking about images. The BIFF file format is simply one way to realize the BIFF image concept.
- Finally, BIFF is the name of a library containing procedures for handling the file format.

All three approaches will be described in this report. Chapter 2 gives a description of the image concept, the "semantics of digital images" so to speak. The description is rather superficial, for a more thorough-going description see [13].

The file format handled in chapter 3 is a slightly modified version of the original format described in [13]. The modification makes it possible to handle larger image files, the previous version was limited to about 32 MB. If chapter 2 gives the "semantics of digital images" then chapter 3 certainly gives the "syntax of digital images".

In chapter 4 you will find a description of the subroutine library developed to handle this file format. The library is written in C for UNIX systems. It is the intention of the author to be so precise that the report can be used as a manual for the library. This report documents version 2 of the library.

In Appendix A you will find some definitions named *blab.h*, making programming in C a cleaner work, thanks to Otto Milvang for these. The definitions are used in the implementation of BIFF.

Appendix B contains the definitions necessary to use the BIFF file format, an include-file *biff.h* with several new definitions.

A previous report from the image processing laboratory [15] documents version 1 of the library. Chapters 2 and 3 describing the image concept and the file format are nearly identical, while chapter 4 is totally rewritten, as is the library.

Chapter 2

The BIFF image concept

The purpose of this chapter is to describe what we mean with a "digital image". This is done by describing the different elements of a digital image, and the possibilities and limitations of this concept. We will start by giving a short description of the digital image, and proceed with a more formal definition of the different elements in the image, building up definitions from the bottom.

Conceptually the digital image contains three main parts, the *info-data*, the *band-data* and the *block-data*. The info-data contains various information about the image, textual as well as numerical. The band-data contains numerical information representing the intensity distributions in the bands, the image may contain any number of bands, numbered from one. The bands have individual sizes and pixel types, sizes are chosen freely and a set of pixel types are defined. Position and magnification of each band is described relative to a global coordinate system. Finally the block-data contains whatever the user wants, no restrictions are placed on it.

2.1 Definitions

- A PIXEL is a basic unit, a variable able to contain information about the intensity of a local area of a picture. A pixel has a certain *type* and thus a certain domain, and a *value* within this domain.

- A BAND is a rectangular arrangement of pixels, all of them having the same type.

- THE LOCAL COORDINATE SYSTEM. To be able to name each individual pixel within a band, a coordinate system local to the band is used. The local coordinate system has two axes, X and Y, the X-index grows from left to right starting with one, ending with *xsize*, and the Y-index grows from top to bottom starting with one and ending with *ysize*. Thus, the upper, left pixel of any band has local coordinates (X,Y) = (1,1), and the bottom, right has coordinates (X,Y) = (xsize,ysize).

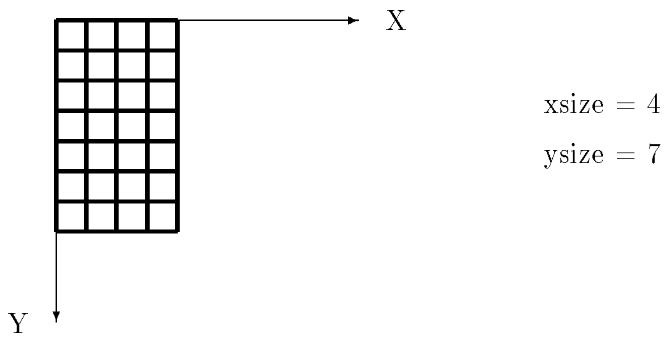
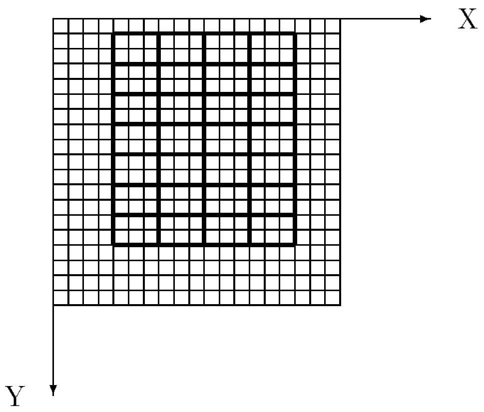


Figure 1 A band in its local coordinate system.

- A LINE is a sequence of pixels from one band, containing all pixels with some fixed Y-coordinate, ordered with growing X-coordinate.
- A COLUMN is a sequence of pixels from one band, containing all pixels with some fixed X-coordinate, ordered with growing Y-coordinate.
- THE GLOBAL COORDINATE SYSTEM. We often need more than one band in an image. Two bands in an image can represent the same spatial area (on the surface of the earth, or inside an engineering drawing), or different spatial areas. They can have identical, or different spatial resolution. To store such information, we use a global coordinate system. For each band, two parameters are used to describe the spatial resolution. *Xmag* is a magnification factor in the X-direction, and *ymag* is a magnification factor in the Y-direction. If each pixel value of a band is repeated xmag times in X-direction and ymag times

in Y-direction, then each of these new pixels will cover the same "true" area as a pixel from another band in the same image, treated the same way.

To describe the relative position of these "blown-up" bands, we use two more parameters, *xstart* and *ystart*. They specify the coordinates in a global coordinate system of the pixel that is numbered (1,1) in the local system. By specifying xstart and ystart of all bands, the relative position of the bands are given. The axes of the global coordinate system should be parallel to the axes of the local systems, and the origin should be chosen so that all bands are located entirely in the positive quadrant. Figure 2 gives an example, the band from figure 1 is shown in a global coordinate system. All six parameters, xsize, ysize, xstart, ystart, xmag and ymag, are positive integers. When there could be confusion between the pixels in the local coordinate system, and the elements in the global coordinate system, the names *locel* and *globel* could be used.



xsize = 4

ysize = 7

xstart = 5

ystart = 2

xmag = 3

ymag = 2

Figure 2 A band located in global coordinate system.

2.2 Info-data

The info-data contains all information necessary to describe the bands, and other information connected to the image.

- TITLE A short text describing the image.
- NBANDS Number of bands in the image.
- PROT-CODE A protection code saying who can do what.
- PARAM Eight user-defined parameters that may contain any sort of information, each with the size of an integer. We recommend that the first parameter is used as an identification of the type of image. The use of the last seven parameters could then be standardized among the users of this particular image type.
- BANDARRAY Numerical information about each band, containing:
  - PIXTYP The pixel type of the band. The pixeltype must be one of the following 12 types: bit, 2 bits, 4 bits, unsigned byte, signed byte, unsigned short, signed short, integer, real, complex, double precision real and double precision complex.
  - XSIZE Horizontal band size.
  - YSIZE Vertical band size.
  - XSTART Horizontal positioning.
  - YSTART Vertical positioning.
  - XMAG Horizontal magnification factor.
  - YMAG Vertical magnification factor.
- NCHARS Number of characters in text-data.
- TEXT-DATA Textual description of the image. The text-data is a sequence of characters of any length, and can even grow as new comments are added.
- NBLOCKS Number of blocks in block-data.

The bands of an image may have different pixeltypes, as well as different sizes, magnifications and locations.

2.3 Band-data

The band-data contains the pixel values in every pixel of every band. There are "nbands" bands, nbands $>= 1$. Thus, the band-data only contains a collection of bands, possibly with different pixel types and sizes, in their individual local coordinate systems. This conceptual image definition does not define any ordering of the pixels.

2.4 Block-data

Block-data is a collection of blocks, each containing 512 bytes. The contents of each block is entirely decided by the user, and the user must self keep in mind what is stored where and how to interpret the data. The blocks can be used to store histograms, color tables etc. It has mainly interest for non-textual data, as info-data can contain any amount of text. There are nblocks blocks, nblocks$>= 0$. The number of blocks can increase, new blocks can be added to the image.

2.5 Conclusion

A main difference between the BIFF image concept and other file formats, is the existence of a *conceptual* definition. Other formats are usually only file formats, the underlying image concept is only given implicit. If we still compare the BIFF image concept with one of the newer such definitions like the file format of the HIPS Image Processing System [17], the main difference is the freedom in BIFF of giving each band a separate size and pixel type. As an example of the flexibility of this concept, the well-known pyramid image structure can be defined as a BIFF-image.

Another important property is the global coordinate system, which is seldom found in other systems. It could have been defined more general, like real values of position and magnification, and allowing rotation of the bands. This would however constitute a more complicated system, and the current definition seems to cover most of the needs.

Finally we should stress the need for text-data of any size which can also be found in several other new formats, like the HIPS system.

BIFF was defined primarily for traditional image processing, and thus the definition was limited to only three dimensions. Time sequences of images, and volume elements images, are increasingly more important in image processing. Larger computers make the processing of larger data sets possible, and both four and five dimensions will be needed in the future. Some existing formats [3] allow any number of dimensions, and this might be a future extension of the BIFF format.

Another possible extension is to include other representation systems, like vector coding, quad trees etc. in addition to raster format. Representation system would then be a property of each band in the image, the first band could be in raster format while the second is vector coded.

Chapter 3

The BIFF image file format

The file format is a concrete implementation of the image concept. Note that it is a partial implementation, it places restrictions on the images that are not inherent in the conceptual image definition.

The three logical parts of the image is, for practical reasons, divided into four physical fields. The *info-field*, the *band-field*, the *text-field* and the *block-field*. Info-data is split into info-field and text-field, otherwise there is a one-to-one correspondence. The ordering on the file is as mentioned here, info – band – text – block. Each of these fields cover a multiple of 512 bytes to simplify file handling in operating systems reading and writing whole "records". In the beginning of the info-field, the sizes of all four fields can be found.

3.1 Info-field

The info-field contains, with a few exceptions, the elements of info-data. One exception is the protection code, a protection code stored in the file is of little use. Instead, the operating system protection is used. In UNIX, three user groups and three permissions are defined. Thus, there are nine protection elements. They are the file owner, the members of the group of the file, and all others – allowed to read the file, write to the file, and execute the file. Other operating systems should use their protection systems.

Another exception is the text-data, which is given a separate field, the text-field. This field is placed after the band-field. This makes it easier and more efficient to enlarge the text-data.

In addition, the info-field contains a code identifying that this is a file following the BIFF format. And finally, as mentioned, the physical sizes of each of the four fields. Even the info-field may occupy more than 512 bytes.

The way these data are ordered and coded in the info-field is described in table 3.1.

From byte	To byte	Length	Datatype	Contents
1	8	8	char	"BIFF-UIO"
9	12	4	int	number of info-blocks
13	16	4	int	number of band-blocks
17	20	4	int	number of text-blocks
21	24	4	int	number of block-blocks
25	56	32	char	image title
33	88	32	–	user-defined parameters
89	92	4	int	number of free characters
93	96	4	int	number of bands
97	100	4	int	pixeltype band 1
101	104	4	int	xsize band 1
105	108	4	int	ysize band 1
109	112	4	int	xstart band 1
113	116	4	int	ystart band 1
117	120	4	int	xmag band 1
121	124	4	int	ymag band 1
125	128	4	–	not used
129	160	32	6*int	band 2, if any, like 97–128
161	192	32	6*int	band 3, if any
and	so	on	if	more bands

Table 3.1: Contents of info field

The char data are coded in ASCII, the int-parameters are coded with most significant bit first, no matter what the machine uses. The library converts if necessary.

3.2 Band-field

The band-field contains the pixel values for all pixels in all bands. They are organized in the samples-lines-bands ordering, samples varies fastest, then lines, and finally bands. This ordering is, to some extent, preserved when an image is read from file into program variables in memory. Thus, when programming an algorithm that is traversing a band, you should let the X-coordinate vary fastest. Each band starts at a 512-byte-boundary on file, but within each band the pixels are packed completely. The bit-coding of each pixel is machine-dependent to avoid conversion every time an image is read or written.

3.3 Text-field

The text-field contains a sequence of nchars characters, without any predefined line-structure. Even this field occupies a multiple of 512 bytes, so there is usually an unused part at the end. If new comments are added to the text-data, it may fit in, or it may not, in which case the field will grow 512 bytes. The characters are coded in ASCII.

3.4 Block-field

Here you will find nblocks elements, each occupying 512 bytes. As pointed out earlier, the system does not make any attempt to interpret these data, you are offered a storage area, nothing else.

3.5 Conclusion

We have chosen to store a whole band as one piece of data, there is no blocking. On small computers without virtual memory this could cause a problem. Even on larger computers, one should try to follow the horizontal lines of the image when possible, to reduce the number of page-faults. The problem must be remembered, but is considered far less than the problem of reading and writing small blocks of the band. If extreme image sizes and small memory resources still requires the image to be split, the blocks can be stored as separate bands in an image, and the global coordinate system can be used to describe the relationship between the bands.

We have also chosen a fixed storage ordering, column varying fastest, then line and finally band. Some formats allow several different ordering systems [16]. This may increase efficiency in a few special algorithms, but makes the file handling somewhat more complicated, and adds nothing to the flexibility from the users (programmers) point of view, the user should not have to be concerned with the file format at all.

Chapter 4

The BIFF procedure library

BIFF is a routine library for handling the Blab Image File Format. All access to the image files will go through the BIFF routine library, which will reduce the programming effort as well as ensure consistency in the image files. The library is written in C for use under the UNIX operating system.

The first implementation of the file format was made for VAX-VMS in 1985 [14], written in Pascal. The next implementation, BIFF V.1 [15], was written in C for UNIX in 1987. This implementation, BIFF V.2, is a complete reorganization of V.1.

Most of the routines in this library are written as functions, they return a function value. When nothing else is explicitly stated, this function value is a status value indicating the success or failure of the routine. What to do with this status value is described in chapter 4.9, error handling.

To avoid naming conflicts, and to simplify identification of library routines, most names of routines, functions, types and macros are initialized with an "I".

The programming syntax used in the following description is plain C modified by the inclusion of biff.h and thus blab.h, see appendices A and B.

All programming examples presume you are using byte-pixels. The necessary changes in case you need other pixel types are described in chapter 4.7.

To use BIFF, you must include <xite/biff.h> in the beginning of your program. The BIFF library is part of a larger library developed at the laboratory. To link with the library, use the command "cxite" which is bundled with the software from the laboratory.

4.1 Data structures

To simplify handling of separate bands as well as whole images, appropriate data types have been defined. Variables of these types are capable of holding respectively a band and a whole image, with pixel data as well as associated information.

4.1.1 The image structure

The type used to represent an image inside a program is naturally called IMAGE. An IMAGE-variable can be un-initialized, or its state can be local, readonly, readwrite or write.

An un-initialized IMAGE-variable has no space for data, it is simply a pointer pointing nowhere. In local state the variable occupies space, and the different elements of the digital image can be read and modified. The name "local" indicates that the data structure is local to the program, it has no connection to any file. The read-only state is used to read *parts* of a BIFF-file. In this state the variable is connected to a BIFF-file, and the desired elements of the image can be read by means of routines in the library. In readwrite-state elements can be read from - as well as written to - an existing BIFF file that the variable is connected to. Finally the write-state can be used to create a new BIFF-file according to the size of the variable that is connected to the file.

The local state is used in nearly all cases. Only when you want to read, write or update minor parts of an image, the other states may be useful.

The IMAGE data structure, when initialized, is a two-headed creature. It contains all information about the image, like the title, number of bands, the text etc., but it also denotes the data itself, a three dimensional collection of pixels. The associated information is read and modified by means of a set of routines according to the principle of data hiding, while the pixels are accessed by means of three indices. The following piece of C-code illustrates how to request the number of bands in the image, and how to index a single pixel.

```
IMAGE i;
int nr_of_bands,first_pix;
< initialize i >
nr_of_bands = Inbands(i); /* read the number of bands */
first_pix = i[1][1][1];   /* read first pixel */
printf(``%d bands, first pixel is %d\n'',nr_of_bands,first_pix);
```

The first index is the band number, starting with 1. The second index is the line index starting on top with 1, and the last index is the column index starting from left with 1. If you omit the last index, you have specified a line, an array of pixels. If you omit the last two indices, you have specified a band.

4.1.2 The band structure

To operate on individual bands of an image, the IBAND type has been declared. A "normal" IBAND variable contains information about the band, as well as the two dimensional collection of pixels. The information can be handled by means of routines in the library, the pixels are manipulated with two indices.

Like the image variable, an un-initialized band variable is only a pointer. After initialization, the state can be normal, subband or absent. A subband does not have its own pixels, but contains references to a part of another band, and can in most cases be handled as if it was a normal band. See section 4.4 for more details. Absent simply means that the pixel data is absent, only the information field is present.

```
IMAGE i;
IBAND b;
int xsize,ysize,first_pix;
< initialize i >
b = i[1]; /* let b point to first band of image i */
xsize = Ixsize(b);
ysize = Iysize(b);
first_pix = b[1][1];
printf(``Xsize %d, Ysize %d\n'',xsize,ysize);
printf(``First pixel is %d\n'',first_pix);
```

We now know enough to program a simple procedure to negate one band into another.

```
PROCEDURE negate(b1,b2)
  IBAND b1,b2;
BEGIN
  int x,y;
  IF ((Ixsize(b1) NE Ixsize(b2))
    OR (Iysize(b1) NE Iysize(b2)))
```

```
  THEN
    /* error message */
  ELSE
    FOR (y=1; y LE Iysize(b1); INC y)
      FOR (x=1; x LE Ixsize(b1); INC x)
        b2[y][x] = 255 - b1[y][x];
      ENDFOR;
    ENDFOR;
  ENDIF;
END;
```

The documentation of the routine should state whether identical in and out parameters are legal, in this case negate(b,b) is legal and will negate the band into itself. Notice the ordering of the loops, let the x index run fastest (inner loop) to reduce the number of page-faults.

4.1.3 Implementation of the data structures

This section describes the actual implementation of the data structures. As the users never have to access the associated information directly, but can manipulate it through a set of routines, this section is totally irrelevant to most users of the library. So skip this at first reading, go immediately to section 4.2.

An un-initialized image-variable is a pointer. After initialization, the pointer points to an array of band-pointers. The number of elements in this array is number-of-bands+1.

Since the first band is band number 1, and C-arrays start with 0, the first pointer is free for other use. It points to a struct containing the relevant information for the picture, such as number of bands, filename (if any), title, image state, etc.

The rest of the pointers in the array points to the individual bands of the image. The band structure is very similar to the image. It contains an array of pointers, the number of elements equals the number of lines plus one. Since the line index starts at 1 while C-arrays start at 0, the first pointer can be used to hold a struct containing relevant information about the band. The rest of the pointers point to the individual lines, or rather one element ahead of the lines, to compensate for the 0-element. The whole structure is illustrated in figure 4.1.

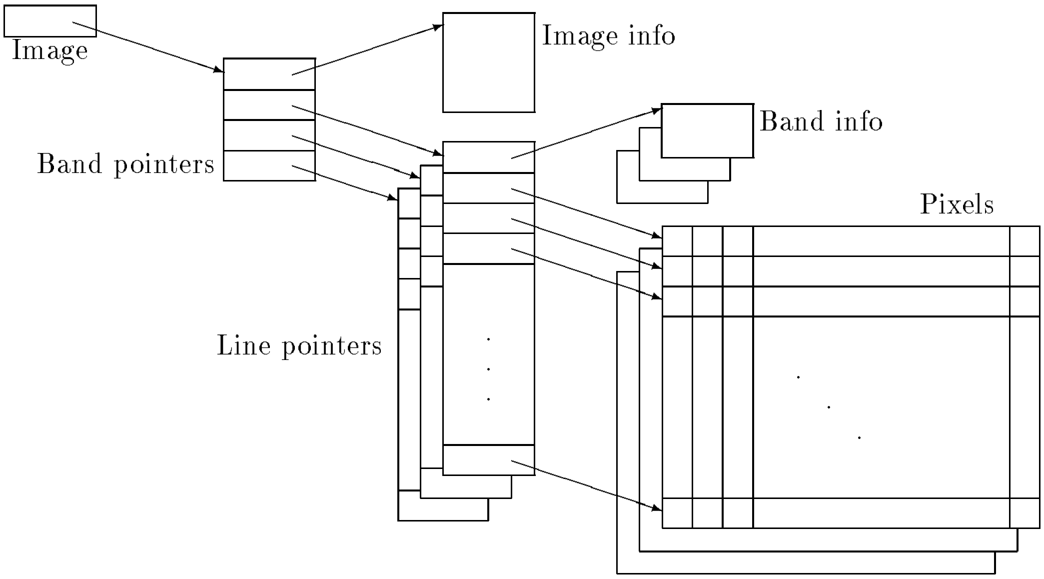


Figure 4.1: The image structure, 3-band image. Pointers in band 2 and 3 are not drawn to avoid complete chaos.

A "normal" band will be allocated so that the lines of the band will be adjacent. Thus, it is possible to traverse a band by placing a pointer at the beginning of the band, and ++'ing along. Routines should however also be able to handle subbands, which are *never* contiguous. The conclusion is that one should always use normal [line][column] indexing. Experience has shown that full indexing in this pointer structure is only marginally slower than one running pointer, especially when you compile with optimization. If speed requirements still demands a running pointer, you should at least test the band state.

The subband structure is nearly identical to the band structure, the difference is that the line pointers point somewhere into the middle of a band, again compensated one position because C arrays start at 0 and pixel indexing at 1. Any number of subbands can be connected to one band. Figure 4.2 illustrates the connection between a band and a subband.

The actual type definitions are found in appendix B.

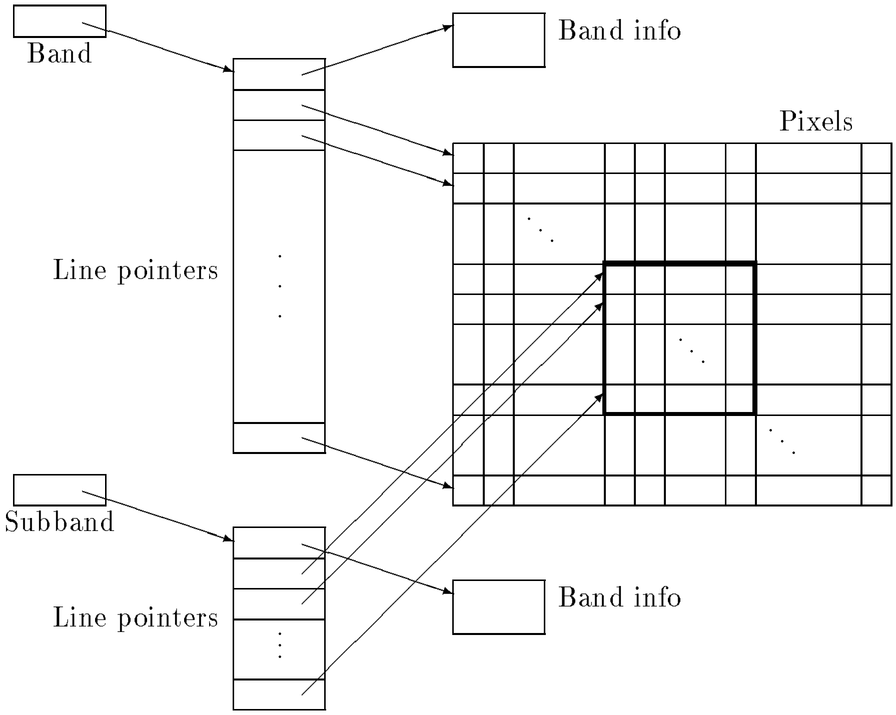


Figure 4.2: The subband structure

4.2 Initialization, input, output and deallocation

This section describes how images can be initialized, how to read a whole image from file, how to write a whole image back to file, and finally how to deallocate memory space used by an image or by a band. The birth, life and death of an image.

4.2.1 Image initialization

A set of routines are at disposal for initializing an image; building up the image structure and assigning initial values to the parameters of the image. The most general way to build the image structure is to start with a call to Init_image, where you give the desired title and number of bands, and then one call to Imake_band for every band, giving pixel type and size.

```
IMAGE i;
int bn,nbands,xsize,ysize;
IPIXTYP pt;
< read nbands >
i = Init_image(nbands,``test image'');
FOR (bn = 1; bn LE nbands; INC bn)
  < read pt, xsize and ysize >
  i[bn] = Imake_band(pt,xsize,ysize);
ENDFOR;
```

If all bands should have the same size and pixel type, you can replace all calls to Imake_band with a single call to Imake_bands :

```
IMAGE i;
int nbands,xsize,ysize;
IPIXTYP pt;
< read nbands, pt, xsize and ysize >
i = Init_image(nbands,``test image'');
Imake_bands(i,pt,xsize,ysize);
```

– or even simpler, a call to Imake_image :

```
IMAGE i;
int nbands,xsize,ysize;
IPIXTYP pt;
< read nbands, pt, xsize and ysize >
i = Imake_image(nbands,``test image'',pt,xsize,ysize);
```

After return from this function, the state of the image is local, and it can be indexed and handled any way.

For special purposes there are also two routines called Init_band and Init_bands that can be called in stead of Imake_band(s). The difference is that the Init_band-routines create "absent" bands, there is no space for pixels.

Often there will be need for creating an image of size identical to an old image. For this purpose the Icopy_init function can be useful. The new image will have the same number of bands as the old one, and every band will have the same size and pixel type as the corresponding band in the old image, but the pixel values are not copied.

```
IMAGE i1,i2;
< Read i1 >
i2 = Icopy_init(i1);
/* i2 is now initialized and state is local */
< Let i2 be a function of i1 >
< Save i2 >
```

4.2.2 Read and write a whole image

The normal start of an image processing program is to read an image from file. Thereafter one can modify it, and write it back to a new file. To read an entire image from file, we use Iread_image.

```
IMAGE i;
char *filename;
i = Iread_image(filename);
```

This call will open the specified image file, examine the size of the image, build up the image structure according to that size, read data from file, and close the file. At return from the routine, the state of the image is local.

A similar routine will do the job of writing an image to file.

```
Iwrite_image(i,filename);
```

The image must be in local state before calling the routine, and is still local after returning from the routine. This means you can continue handling the image, and write it to another file later on. The named file must not exist before calling the routine, an attempt to overwrite an old file is considered an error for security reasons. Also for security, a "readers-writers" file protection system is implemented which assures that when you write an image file, you have exclusive access to the file. No one else can read or write as long as you use it. However, any number of users can read an image file simultaneously.

If you call Iread_image or Iwrite_image with the empty string as filename, the routines will read from, respectively write to, standard input and standard output. This means images can be piped between programs, a feature that has shown to be very useful [17], [18]. You can also use "-n" as filename, n being a number, in which case n is interpreted as the number of an open channel to read from, or write to.

The Iread_image and Iwrite_image routines does not read and write block-data. Separate routines can do this job for you, see section 4.8.

4.2.3 Deallocation of band and image

If your program handles lots of images, and the virtual address space is limited, it can be recommended to deallocate address space when there is no need for it any more. To deallocate a band you call Idel_band.

```
IBAND b;
IPIXTYP pt;
int xsize,ysize;
b = Imake_band(pt,xsize,ysize);
<use b>
Idel_band(&b);
```

To deallocate an entire image, there is a corresponding Idel_image routine.

```
IMAGE i;
IPIXTYP pt;
int xsize,ysize;
i = Imake_image(nbands,''Title'',pt,xsize,ysize);
<use i>
Idel_image(&i);
```

4.3 Information handling

The information connected to the entire image, and the information associated with each band, can be obtained and modified by means of a set of routines.

4.3.1 Image information handling

The following routines are used for handling the information associated with an entire image.

- Istatus(i) returns the state of image i, one of the following predefined values: Ilocal, Ireadonly, Ireadwrite, Iwrite.
- Inbands(i) returns the number of bands in the image i.
- Inchars(i) returns the number of characters in the text field.

- Ititle(i) returns a pointer to a copy of the title of the image.
- Iset_title(i,title) changes the title of the image.
- Iset_nbands(i,nbands) changes the number of bands in the image. If you decrease the number of bands, the removed ones are really gone and can not be accessed. If you increase, the new bands are uninitialized.
- Iget_param(i,parmnr,parmval) returns one of the eight user defined parameters, number "parmnr".
- Iset_param(i,parmnr,parmval) modifies the specified user defined parameter.

4.3.2 Band information handling

The purpose of these routines is to handle the information associated with a single band.

- Iband_status(b) returns the state of band b, one of: Inormal, Isubband, Iabsent.
- Ixsize(b) returns the horizontal size of the band.
- Iysize(b) returns the vertical size of the band.
- Ixstart(b) returns the horizontal startpoint of the band. Startpoint-values are initially 1.
- Iystart(b) returns the vertical startpoint of the band.
- Ixmag(b) returns the horizontal magnification factor of the band. Magnification factors are initially 1.
- Iymag(b) returns the vertical magnification factor of the band.
- Iset_start(b,xstart,ystart) changes the startpoint values.
- Iset_mag(b,xmag,ymag) changes the magnification factors.
- Iget_roi(b,xstart,ystart,xsize,ysize) returns region-of-interest parameters. The region-of-interest routines simply provide a way to store and retrieve information while handling a band, the values are not used or interpreted by the system. Initial values specify the whole band.

- Iset_roi(b,xstart,ystart,xsize,ysize) changes the region-of-interest parameters.
- Ipixtyp(b) returns the pixel type of band b. See below for available pixel types.
- Ipixsize(pt) returns the size of a pixel in bits, given a pixel type.
- Ibandsize(b) returns the size of the band in bytes, counting only the space occupied by the pixels, not the pointer structure.

The type IPIXTYP has been defined, and the Ipixtyp-routine will return a value of that type. The following IPIXTYP-values are defined: Iu_byte_typ, Is_byte_typ, Iu_short_typ, Is_short_typ, Integer_typ, Ireal_typ, Icomplex_typ, Idouble_typ and Id_complex_typ. See also section 4.7 concerning the use of different pixel types.

4.4 Subimages and subbands

There is sometimes a need to operate on parts of an image. Instead of passing along to a routine a large image plus information that describes what parts of the image should be processed, you may create subimages and subbands.

The subimage mechanism is very simple to use. Let us assume that the negate-routine defined in section 4.1.2 expected two IMAGE-parameters in stead of IBAND-parameters. We have a 3-band image on file, but we only want to negate the 2. band.

```
IMAGE i1,i2;
char *filename;
i1 = Iread_image(filename);
i2 = Init_image(1,``Subimage'');
i2[1] = i1[2];
negate(i2,i2);
/* The 2. band of i1 is now negated */
```

The subband is somewhat more complicated, it needs an extra routine for establishing the data structure. Let us return to the original band-version of negate. We have a 1-band image, and we would like to negate the upper, left quadrant only.

```
IMAGE i;
IBAND b;
char *filename;
i = Iread_image(filename);
b = Imake_subband(i[1],1,1,Ixsize(i[1])/2,Iysize(i[1])/2);
negate(b,b);
/* The upper, left quadrant of i[1] is now negated */
```

If we let another subband point to the lower, right quadrant, then we can negate the upper, left part of the band into the lower, right. However, watch out for partly overlapping subbands, the effect can be hard to predict.

The subimage and subband possibilities may of course be mixed. The different bands of an image variable can point to its own bands, to bands in other images or subbands in other images, in any order. Even subbands of subbands can be created. If a band with subband-pointers to it is deallocated, the subbands can not be accessed. Deallocation of subbands have no influence on the "main"-band. When you write a subband to file, the pixel values are simply copied. The connection between the bands is not represented in the files.

4.5 Bit-manipulation

In order to manipulate a pixel bit by bit, four routines have been defined. This feature is only implemented for byte-pixels, and the routines are defined as macros for optimal efficiency.

- Igetbit(b,y,x,bit) returns the bit found in band b, line y, column x, bit number bit. Bit nr 1 is the least significant bit, bit nr 8 is the most significant bit.
- Isetbit(b,y,x,bit) sets the bit to 1.
- Iclearbit(b,y,x,bit) clears the described bit.
- Inegbit(b,y,x,bit) negates the bit.

By means of these tools, a byte-band can be used as eight bitplanes.

4.6 Text handling

Conceptually, the text field is one long sequence of characters with an associated "cursor", like the UNIX definition of files. The textfield can be manipulated completely, characters and strings can be read, inserted and deleted. Initially the cursor is placed at the first character, in position one. It can at most move one position past the last character.

The "low-level" routines constitute a basis, anything can be done by means of that basis. The "high-level" routines are built upon the low-level, and simply reduces the effort of doing larger operations on the text data.

4.6.1 Low level

- Igetpos(i) returns the cursor position of image i.
- Isetpos(i,pos) puts the cursor in the specified position.
- Igetchar(i,c) fetches the character at the cursor, and moves one position ahead.
- Iputchar(i,c) places the character c at the cursor, and moves ahead. Old text is overwritten.
- Idel_text(i) deletes all text from cursor to end of text.

4.6.2 High level

- Iput_text(i,s) inserts the text s at cursor position. Old text is overwritten, and cursor is placed right after the new text.
- Iget_line(i,s) fetches a line from the textfield. A line will stop at C-J (Control-J), or after the 80. character if no C-J is encountered.
- Iappend_line(i,s) will append the text s to the end of the textfield, and finally add a C-J.
- Itype_text(i,fp) will copy the whole text field to the file identified by file pointer fp.
- Iedit_text(i) lets you edit the text field. The text is copied to a temporary file, the editor specified by environment variable EDITOR is activated, and when you save the result and exit from the editor, the new text is put into the image again, overwriting the old text.

- Iend_of_text(i) reports whether the cursor is at end-of-text.
- Icopy_text(from,to) appends the text found in from-image to the text in to-image.
- Ihistory(i,prog,comment) appends a line consisting of three fields, 1) the current time and date, 2) the program name supplied as the prog-parameter – use argv[0] – and finally 3) a free comment on what has been done to the image.

4.7 Available pixel types

There are currently 12 pixel types in the conceptual definition of BIFF, but only 8 have been implemented here, 1 bit, 2 bits and 4 bits are missing.

```
TYPENAME      EXPLANATION

UNS_BYTE      unsigned byte
SIGN_BYTE     signed byte
UNS_SHORT     unsigned 16bit integer
SIGN_SHORT    signed 16bit integer
INTEGER       signed 32bit integer
REAL          32bit floating point
COMPLEX       2*32bit floating point
DOUBLE        64bit floating point
DCOMPLEX      2*64bit floating point
```

To be able to generate the right code for indexing the bands, the compiler has to know the pixel type of the band. Several band-types have thus been defined.

```
typedef UNS_BYTE**    IBAND;
typedef UNS_BYTE**    IUB_BAND;
typedef SIGN_BYTE**   ISB_BAND;
typedef UNS_SHORT**   IUS_BAND;
typedef SIGN_SHORT**  ISS_BAND;
typedef INTEGER**     II_BAND;
typedef REAL**        IR_BAND;
typedef COMPLEX**     IC_BAND;
typedef DOUBLE**      ID_BAND;
typedef DCOMPLEX**    IDC_BAND;
```

Finally the following image-types are defined:

```
typedef UNS_BYTE***   IMAGE;
typedef UNS_BYTE***   IUB_IMAGE;
typedef SIGN_BYTE***  ISB_IMAGE;
typedef UNS_SHORT***  IUS_IMAGE;
typedef SIGN_SHORT*** ISS_IMAGE;
typedef INTEGER***    II_IMAGE;
typedef REAL***       IR_IMAGE;
typedef COMPLEX***    IC_IMAGE;
typedef DOUBLE***     ID_IMAGE;
typedef DCOMPLEX***   IDC_IMAGE;
```

If you are handling an image where all bands have the same pixeltype, the image can be declared as the corresponding type, and you can use the normal [band][line][column] image–indexing. An example: integer pixels.

```
II_IMAGE i;
i = Iread_image(``filename'');
< Verify that really INTEGER pixel type >
i[1][1][1] = 1763957; /* can now assign integer values */
```

If, however, the image consists of bands with different pixel types, a little more care must be taken, you should only use the [line][column] band–indexing. Assuming first band is INTEGER and second is REAL, the following code can be used.

```
IMAGE i;
II_BAND ib;
IR_BAND rb;
i = Iread_image(``filename'');
< Verify that first band INTEGER and second REAL >
ib = (II_BAND)i[1];
rb = (IR_BAND)i[2];
ib[1][1] = 1763957;
rb[1][1] = 15.497;
```

You may also let the type of the image correspond to one of the occurring pixel types, and use image-indexing on the bands of that pixel type:

```
II_IMAGE i;
IR_BAND rb;
i = Iread_image(''filename'');
< Verify that first band INTEGER and second REAL >
rb = (IR_BAND)i[2];
i[1][1][1] = 1763957;
rb[1][1] = 15.497;
```

The following constants have been defined to identify the pixel types of different bands:

- Iu_byte_typ
- Is_byte_typ
- Iu_short_typ
- Is_short_typ
- Integer_typ
- Ireal_typ
- Icomplex_typ
- Idouble_typ
- Id_complex_typ

– and to verify that a band b has integer pixels, you simply test

```
IF (Ipixtyp(b) EQ Integer_typ) THEN ....
```

These constants are also used to indicate what pixel type you want in new images. We are now creating a 2-band byte-image.

```
IMAGE i;
int xsize,ysize;
IPIXTYP pt;
pt = Iu_byte_typ;
xsize = 500;
ysize = 700;
i = Imake_image(2,''test image'',pt,xsize,ysize);
```

4.8 Readonly, readwrite and write state

If you are interested in reading only minor parts of an image like one of the bands, or only the text, it is not necessary to read the entire image with Iread_image. Special routines let you connect to an image file and read or write selected parts of the image. The image is declared as usual, but then we have to connect the image to an image file. This is done with a call to Iopen_image, where we specify one of the three remaining state possibilities of the image. The four defined constants are:

- Ilocal
- Ireadonly
- Ireadwrite
- Iwrite

Ireadonly is naturally used to read parts of the image file. Ireadwrite is used to read and write, in any sequence and combination. We can read a band, modify it, and write it back, we can read one band and write another, or whatever. Just notice that the sizes of the bands to be written must correspond with the bands on file. Finally Iwrite is used to create a new image file. Using Ireadonly or Ireadwrite all image sizes are read from file at opening, and the image parameter sent along should be un-initialized. Using Iwrite, the image must be initialized before calling Iopen_image, to specify the size of the image. Let us try to read the first band of an image.

```
IMAGE i;
char *filename;
Iopen_image(&i,filename,Ireadonly);
  /* Image state is Ireadonly */
Iread_band(i,1);
  /* Band 1 has been read */
Iclose_image(i);
  /* State is Ilocal */
```

We can now handle the image in local state. Even the sizes of other bands – if any – can be examined, but the pixels can not be indexed. These bands have state absent. The following band state constants are defined:

- Inormal
- Isubband
- Iabsent

In addition to Iread_band there are two other routines that can be used in the readonly situation.

- Iread_text(i) reads the entire text field into the image i.
- Iread_block(i,blk,blnr) reads block blnr of the block field into the buffer blk.

In Ireadwrite state, the previous read-routines can be used, in addition to corresponding write-routines:

- Iwrite_band(i,n) writes band n of image i.
- Iwrite_text(i) writes the text in image i to file.
- Iwrite_block(i,blk,blnr) writes the 512-byte buffer blk to position blnr on file.

The Iread_block and Iwrite_block routines are the only way to read and write block-data, this field is not read/written automatically with Iread_image and Iwrite_image.

To create a new image file, the IMAGE-variable must first be initialized by means of the routines described earlier; Imake_image, Init_image, Imake_band, Imake_bands, Init_band or Init_bands. The image is then local, and can be connected to a new file:

```
IMAGE i;
char *filename;
char *title;
int nb,xsize,ysize;
IPIXTYP pt;
< init filename, title, nb, pt, xsize & ysize >
i = Imake_image(nb,title,pt,xsize,ysize);
  /* State is local */
Iopen_image(&i,filename,Iwrite);
  /* State is write */
```

```
Iwrite_band(i,1);
  /* Band 1 is written */
Iclose_image(i);
  /* State is local again */
```

Iclose_image will close the file again, and break the connection between the file and the image-variable. The variable is then in local state. If you initialize an image with a certain number of bands, open a new image file, but only write some of the bands (or none) before closing, then there will be logically allocated space for all the bands on file. All bands can later be read, all "missing" values will read zero. An image in write state can use the read- as well as the write-routines, like when state is readwrite.

4.9 Error handling

Most routines in the library return a status value. To verify the success of a routine, simply test that value.

```
IF (Iwrite_image(i,filename) EQ Iok) THEN .. /* success */
```

In stead of testing the status value, you can let the system performe error handling. The two error handling services offered by the system are automatic error message display, and job abortion. Error message display means that an error message is sent to standard error when an error is encountered in any of the routines. Job abortion means that the whole process is terminated, eventually after sending the error message. The two mechanisms can be switched on and off.

- Ierr_message(ernr) returns the error message corresponding to an error number returned from a routine.
- Ierr_action(ernr) performs error handling; displays error message corresponding to the error number if that system is turned on, and aborts the job if that system is turned on.
- Iset_message(on) turns the automatic error message display system on when the parameter is TRUE, off when FALSE is given.
- Iset_abort(on) turns the abort-job-on-error system on or off depending on the logical value of the parameter.

Both error handling mechanisms are initially turned off.

The possible error numbers are listed in Appendix B.

4.10 Conclusion

A complete set of routines for handling the BIFF image file format is presented. It is complete in the sense that any part of the image can be read from file, updated on the file, or written to a new file.

The package has been implemented in accordance with object-oriented principles, only system routines should access the data and hide the actual implementation of the data structures. This simplifies the use of the file format, simplifies maintenance of the package, and ensures consistency in image files.

Ease of use has been important for the design. The two most important routines, read and write an image, are made as simple as possible, only the image variable and the image file name is involved. This is to be compared with the I/O routines proposed in [9], [19], [20], [21] and [22], where corresponding operations require several routine calls as well as long parameter lists. These proposals aim at being operating system independent. The BIFF library is written for use under the UNIX operating system, but uses only simple system calls like read, write, malloc and free, which could be implemented in most other operating systems. Thus, it should be quite trivial to adapt the library to another operating system.

The combination of pixel data and associated information into one data structure, simplifies the use significantly. Parameter lists of image processing routines will only contain the bands or images to be processed, actual sizes and pixel types need no separate parameters. Similar possibilities seems to be missing in other systems.

A unique property of the library is the sub-band and sub-image mechanism, allowing simple processing of parts of an image. The image processing routines need not be aware of the fact that only parts of the image is processed, loops always traverse what seems to be the whole image.

Chapter 5

Summary and conclusion

A raster image concept, a raster image file format, and a routine library with data structures for handling these raster images, are presented. The raster image concept defines images to be a sequence of bands of individual size and pixeltype, organized in a common coordinate system. Additional information, numerical as well as textual, can also be a part of the image.

The image file format is a straightforward ordering of these data on file. A header with essential information is followed by the pixel values stored line by line, band by band. At the end, any amount of textual and numerical information can be added.

The routine library is written in C for UNIX systems. The library includes type definitions for images and bands, and more than 60 routines for handling these data. The data objects contain the individual pixel values, as well as the associated information. Thus, no extra size parameter is required when transferring an image to a routine. High-level routines let you read and write whole images, low-level routines allow you to read, write or update minor parts of the image. Different routines are defined to manipulate the information fields, while the pixels are manipulated by normal indexing. Sub-images and sub-bands can be defined for simple processing of parts of an image.

The image concept and file format has shown its usefulness through daily use since 1985, and so far it seems to have covered the needs of the diverse users of the image processing lab. The routine library is a product of 1988, but it has already been tested through several applications made by several programmers. My hope is that the definitions and the library will be found useful even outside the laboratory.

Appendix A

blab.h

Here follows a set of definitions making programming C, and especially reading C, a cleaner work. They are written by Otto Milvang.

```
/* ---------------------------------------------------- */
/*                                                      */
/*      Module  : blab.h                                */
/*      Author  : Otto Milvang                          */
/*      Date    : 23-Jul-1986                           */
/*      Version : 1.0                                   */
/*                                                      */
/* ---------------------------------------------------- */

#ifndef         BLABDEF
#define         BLABDEF         1

#define         FALSE 0
#define         TRUE            1

#define         AND             &&
#define         OR              ||
#define         NOT             !

#define         EQ              ==
#define         NE              !=
#define         LT              <
#define         LE              <=
```

```
#define         GT              >
#define         GE              >=

#define         BAND            &
#define         BOR             |
#define         BNOT            ~
#define         BXOR            ^
#define         LSHF            <<
#define         RSHF            >>

#define         INC             ++
#define         DEC             --
#define         MOD             %
#define         SIZEOF          sizeof

#define         BEGIN           {
#define         END             ;}

#define         IF(e)           { if(e)
#define         THEN            {
#define         ELSEIF(e)       ;} else if(e)
#define         ELSE            ;} else {
#define         ENDIF           ;}}

#define         SWITCH(e)       { switch(e) {
#define         CASE(e)         case e: {
#define         DEFCASE         default: {
#define         ENDCASE         ;} break;
#define         ENDSWITCH       ;}}

#define         FOR(e)          { for(e) {
#define         ENDFOR          ;}}

#define         LOOP            { while(TRUE) {
#define         EXITIF(e)       ; if(e) break;
#define         NEXT            ; continue;
#define         ENDLOOP         ;}}

#define         WHILE(e)        { while(e) {
```

```
#define         ENDWHILE        ;}}

#define         PROGRAM
#define         PROCEDURE       void
#define         FUNCTION

#endif
```

Appendix B

biff.h

These definitions must be included in order to use the BIFF routines.

```
/*============================================================*/
/*  MODULE-NAME  : biff.h - Blab-UiO Image File Format      */
/*  PURPOSE      : DEFINITIONS FOR HANDLING THE BLAB-UIO    */
/*                 IMAGE FILE FORMAT.                       */
/*  WRITTEN BY   : TOR LONNESTAD                            */
/*  VERSION      : 2                                        */
/*  REVISED      : SVEIN BOE                                */
/*  UPDATE       : 95/02/06                                 */
/*  $Id: appendixb.tex,v 1.1 1995/10/03 16:03:43 svein Exp $      */
/*============================================================*/

/*======================================================================*/
/*=========================   INCLUSIONS   ===========================*/
/*======================================================================*/

#ifndef _BIFF_H_
#define _BIFF_H_ 1
#ifdef MSDOS
# ifndef ANSI
#  define ANSI 1
# endif
#endif

#if defined(XITE_HEADERS)
```

```
# include <xite/includes.h>
#endif

#ifndef _XITE_INCLUDES_H_
# if defined(ANSI) && !defined(__STDC__)
#   define __STDC__
# endif

# if defined(__STDC__) && !defined(ANSI)
#   define ANSI
# endif
#endif

#if defined(_XITE_INCLUDES_H_)
# if defined(__STDC__)
#   include XITE_STDIO_H
# endif
# include XITE_LIMITS_H    /* For pixel type definitions */
#else
# if defined(__STDC__)
#   include <stdio.h>
# endif
# include <limits.h>       /* For pixel type definitions */
#endif

/*============================================================*/
/*====================   MACRO FUNCTIONS   ==================*/
/*============================================================*/

/* For all functions, "bit" must be 1..8 */
#define Igetbit(b,y,x,bit) ((b[y][x] BAND Ibitmask[bit])?1:0)
#define Isetbit(b,y,x,bit) (b[y][x] = (b[y][x] BOR Ibitmask[bit]))
#define Iclearbit(b,y,x,bit) (b[y][x] = (b[y][x] BAND Inegbitmask[bit]))
#define Inegbit(b,y,x,bit) (b[y][x] = (b[y][x] BXOR Ibitmask[bit]))

/*============================================================*/
/*================   CONSTANT DEFINITIONS   =================*/
/*============================================================*/
```

```
/*=====   PIXEL TYPE NUMBERS   =====*/
/* When a band is saved to file from a little-endian computer, an extra
   bit is set (pixtyp | Ilittle_endian_mask) */

/*=======   PIXEL TYPE NUMBERS   =======*/
#define Iunknown_typ   -1
#define Ibit_typ        0
#define Ibit2_typ       1
#define Inibble_typ     2
#define Iu_byte_typ     3
#define Is_byte_typ     4
#define Iu_short_typ    5
#define Is_short_typ    6
#define Integer_typ     7
#define Ireal_typ       8
#define Icomplex_typ    9
#define Idouble_typ    10
#define Id_complex_typ 11

/*=======   IMAGE STATES   =======*/
#define Ireadonly      0
#define Ireadwrite     1
#define Iwrite         2
#define Ilocal         3

/*=======   BAND STATES   =======*/
#define Inormal        0
#define Isubband       1
#define Iabsent        2

/*=======   ERROR NUMBERS   =======*/
#define Iok            0
#define Ibifferr       1
#define Iopenerr       2
#define Icloserr       3
#define Inotoperr      4
#define Ibandnrerr     5
#define Isizerr        6
#define Istrtmagerr    7
```

```
#define Ireaderr       8
#define Iwriterr       9
#define Itexterr      10
#define Iblockerr     11
#define Ipixtyperr    12
#define Ilockerr      13
#define Istatuserr    14
#define Iblocknrerr   15
#define Isetposerr    16
#define Igetcharerr   17
#define Infoerr       18
#define Iparnrerr     19
#define Imkarrerr     20
#define Itmpfilerr    21
#define Iroierr       22
#define Imageerr      23
#define Ibanderr      24
#define Imallocerr    25
#define Inotimplerr   99

/*======================================================================*/
/*======================  TYPEDEFINITIONS  ===========================*/
/*======================================================================*/

typedef long IPIXTYP;              /* 0 <= abs(IPIXTYP) <= 11 */
typedef int  BiffStatus;

/* typedef BIT */
/* typedef BIT2 */
/* typedef NIBBLE */
typedef unsigned char UNS_BYTE;
#if CHAR_MAX == SCHAR_MAX
  typedef char SIGN_BYTE;
#else
  typedef signed char SIGN_BYTE;
#endif
typedef unsigned short UNS_SHORT;
typedef short SIGN_SHORT;
#if LONG_MAX > 2147483647
```

```
  typedef int INTEGER;
#else
  typedef long INTEGER;
#endif
typedef float REAL;
typedef struct { float re, im; } COMPLEX;
typedef struct { double re, im; } DCOMPLEX;
typedef double DOUBLE;

typedef UNS_BYTE**    IBAND;
typedef UNS_BYTE**    IUB_BAND;
typedef SIGN_BYTE**   ISB_BAND;
typedef UNS_SHORT**   IUS_BAND;
typedef SIGN_SHORT**  ISS_BAND;
typedef INTEGER**     II_BAND;
typedef REAL**        IR_BAND;
typedef COMPLEX**     IC_BAND;
typedef DOUBLE**      ID_BAND;
typedef DCOMPLEX**    IDC_BAND;

typedef UNS_BYTE***   IMAGE;
typedef UNS_BYTE***   IUB_IMAGE;
typedef SIGN_BYTE***  ISB_IMAGE;
typedef UNS_SHORT***  IUS_IMAGE;
typedef SIGN_SHORT*** ISS_IMAGE;
typedef INTEGER***    II_IMAGE;
typedef REAL***       IR_IMAGE;
typedef COMPLEX***    IC_IMAGE;
typedef DOUBLE***     ID_IMAGE;
typedef DCOMPLEX***   IDC_IMAGE;

typedef struct
{
  /* When a band is saved to file from a little-endian computer, an extra
     bit of pixtyp is set (pixtyp | Ilittle_endian_mask).
     If the band is color, an extra bit is set (pixtyp | Icolor_mask).
     Only the bits not concerned with byte-sex or color are used in
     determining pixel type, pixel size and pixel name. */
```

```
   IPIXTYP pixtyp;
   long xsize;
   long ysize;
   long xstart;
   long ystart;
   long xmag;
   long ymag;
   long status;
   long roi_xstart;
   long roi_ystart;
   long roi_xsize;
   long roi_ysize;
   long* this_struct;
} _Iband_info;

typedef struct
{
   long nbands;
   char *filename;
   char title[36]; /* 33 + alignment */
   long status;
   long param[8];
   long nchars;
   long nblocks;
   long pos;
   char *text;
   long textbufblks;
   long ninfoblks;
   long nbandblks;
   long ntextblks;
   int fd;
} _Info;

#endif
```

Bibliography

[1] L. G. Shapiro. "Data structures for Picture Processing: A Survey" *Computer Graphics and Image Processing*, Vol. 11, 1979, pp. 162–184.

[2] C. T. Zahn, Jr. "Data Structures for Pattern Recognition Algorithms" In:*Data Structures, Computer Graphics, and Pattern Recognition*, Edited by A. Klinger, K. S. Fu and T. L. Kunii. Academic Press, New York, 1977.

[3] D. C. Wells, E. W. Greisen and R. H. Harten. "FITZ: A Flexible Image Transport System" *Astronomy and Astrophysics Supplement Series 44*, 1981, pp. 363–370.

[4] "Yellow Book, Volume VII – Fascicle VII.2, Telegraph and Telematic Sevices Terminal Equipment, Recommendations of the S and T series" *CCITT, VIIth plenary assembly*, Geneva, 10-21 november 1980. ISBN 92-61-01141-1.

[5] *ILIAD User's Guide* IMTEC Image Technology AB, Uppsala, Sweden, 1986.

[6] *I^2S Document Set, System 600* International Imaging Systems, 1500 Buckey Drive, Milpitas, California.

[7] T. Andreassen. "Description of the Program System for Microdensitometry of HRTS Spectral Plates" *HRTS Report No. 3*, Institute for Theoretical Astrophysics, University of Oslo, Norway, 1985.

[8] B. T. Wada. "A Virtual Memory System for Picture Processing" *Communications of the ACM*, Vol. 27, No. 5, May 1984, pp. 444-454.

[9] R. M. Haralik. "Image Access Protocol for Image Processing Software" *IEEE Transactions on Software Engineering*, March 1977, pp. 190 – 192.

[10] R. M. Haralik and G. Minden. "KANDIDATS: An Interactive Image Processing System" *Computer Graphics and Image Processing*, Vol. 8, 1978, pp. 1–15.

[11] B. Morland. "Kjerne for Bildebehandlings-system", (In Norwegian) Institutt for Databehandling, Norges Tekniske Høgskole, Trondheim, Norway, 1981.

[12] W. B. Green. "Digital Image Processing", Chapters 6 and 7. Van Nostrand, New York, 1983.

[13] T. Lønnestad. "Digitale bilete ved Bilethandsamingslaboratoriet" (In Norwegian) *Report No. 3*, Image Processing Laboratory, Department of Informatics, University of Oslo, Norway, 1985. ISBN 82-7476-003-4.

[14] T. Lønnestad. "Programmeringsverkty for handsaming av digitale bilete" (In Norwegian) *Report No.6*, Image Processing Laboratory, Department of Informatics, University of Oslo, Norway, 1985. ISBN 82-7476-006-9.

[15] T. Lønnestad. "BIFF Blab Image File Format" *Report No.14*, Image Processing Laboratory, Department of Informatics, University of Oslo, Norway, 1987. ISBN 82-7476-015-8.

[16] M. Zachrisen. "TIGER-C Raster Graphic System, User's Manual", RUNIT, Trondheim, Norway, 1983.

[17] M. S. Landy, Y. Cohen and G. Sperling. "HIPS: A Unix-Based Image Processing System" *Computer Vision, Graphics and Image Processing*, Vol. 25, 1984, pp. 331–347.

[18] W. R. Stevens and B. R. Hunt. "Software Pipelines in Image Processing" *Computer Graphics and Image Processing*, Vol. 20, 1982, pp. 90–95.

[19] R. G. Hamlet and A. Rosenfeld. "Transportable image-processing Software" *Proc. of National Computer Conference*, June 1979, AFIPS Press, pp. 267–272.

[20] R. G. Hamlet and R. M. Haralick. "Transportablee Package Software", *Software – Practice and Experience*, Vol. 10, 1980, pp. 1009–1027.

[21] S. Krusemark and R. M. Haralick. "Achieving Portability in Image Processing Software Packages" *Proc. of 82 IEEE Computer Society Conference on Pattern Recognition and Image Analysis*, 1982.

[22] S. Krusemark and R. M. Haralick. "An Operating System Interface for Transportable Image Processing Software", *Computer Vision, Graphics and Image Processing*, Vol. 23, 1983, pp. 42–66.