EINDHOVEN UNIVERSITY OF TECHNOLOGY

AIFC sound files decoding for compression

Citation for published version (APA):
Caloz, C. (1993). *AIFC sound files decoding for compression*. (IPO rapport; Vol. 944). Instituut voor Perceptie Onderzoek (IPO).

Document status and date:
Published: 12/11/1993

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 22. Mar. 2023

Institute for Perception Research
PO Box 513, 5600 MB Eindhoven

12.11.1993

Rapport no. 944

AIFC sound files decoding for compression

C. Caloz

Summary

At the I.P.O. research is being done to adapt graphical user interfaces for visually disabled users. In this context, an alternative input/output device, using non-speech/speech audio, si being realised. This involves using sound files whose size is, in general, very large. Consequently, there is a need to compress them. Several compression technics could apply. For example, the DCC algorithms, developed in the Philips Laboratories, are suitable to this task.

The sound files available are binary AIFC files. So, they cannot just be read as text files. They first have to be decoded. On the other hand, these files are created on the IRIS INDIGO system whereas we are working on SUN stations. This will necessitate appropriate conversions of formats. We propose a program that achieves the decoding and converting of the AIFC files data and store them into C++ objects. These objects provide an efficient platform to manipulate the data for compressing. The separation of the program in several files (interfaces and implementations) is a contributing factor to portability and the programming in regard to the object oriented philosophy facilitates to a large extent further developments.

Contents

1 Introduction p1

2 Formal specifications p2

3 General organisation of the program p3

4 Object data dictionary p4

5 Description of the defined functions p7

5.1 retSize() p7

5.2 fretrieve() p7

5.3 print() p7

5.4 Common::getnumChannels() p7

5.5 Common::getnumSampleFrames() p8

5.6 Common::getsampleSize p8

5.7 Common::getsampleRate() p8

6 Access to the sound data p9

7 Conclusions and recommendations p10

1 Introduction

The sound files (see annex) to be used are created on the IRIS INDIGO system and have the so called AIFC format. They are sampled at a DAT quality, that is to say at 48 kbytes/sec. This means that if we produce three seconds of sound, the size of the sound file will already exceed 100 kbytes. So, we see that the sound files are usually very large and that there is a need to compress them. In order to achieve compression, using DCC technics developed by Philips, we first need to get the data from the sound files and to store them in an efficient way. But the sound files have a binary form and cannot simply be read as text files. They have to be decoded. Besides the AIFC format is a Macintosh sound format, depending on the Motorola 68'000 processor, and the sound files have to be used on the SUN system, having an AT&T processor. Consequently, the reading of AIFC sound files on a SUN necessitates conversions of formats. For these purposes we realised a program, written in C++ that provides facilities, in accordance with the object oriented philosophy, to manipulate the data read from the AIFC sound files in order to achieve compression.

2 Formal specifications

In order to prepare the AIFC files for compression, the program should comply with the following :

- All the data from the AIFC sound files have to be read. Consequently, the appropriate formats conversions (from INDIGO to SUN) have to be done.

- It must be possible to decode any AIFC file from the command line according to the following format : <...> executable_filename target_filename.

- Easy access to any data in the read file must be provided. Particularly, it must be possible to get any sound data both in a sequential and in a structured way, that is to say by specifying the numbers of the channel and the byte of the currently visited frame.

- If an error occurs while manipulating the files, the program has to be terminated and a corresponding message has to be displayed.

- The objects containing the data must have the same fields as the actual fields of the AIFC files (see annex 1) in order to facilitate the understanding of the program and to be in perfect accorance with the structure of the input format.

- All these data have to be stored into clear and efficient C++ objects in order to make the program more portable and to facilitate further modifications or extensions.

- The program must provide possibilities for further developments. For example, it should be easy to create an output (compressed) file by specifying its name as a third argument in the command line, the list of errors must also contain errors which could occur when manipulating this output file, if the chunks (see annex 1) of the file would appear in another order as the standard one (which is not excluded), it should be possible to achieve decoding all the same, etc.

3 General organisation of the program

The program is written in the C++ programming language. It physically consists of three separated files :

1) Sound_definitions.h
2) Sound_functions.c
3) Sound_main.c

1) Sound_definitions.h is a header file defining an interface that contains the definitions of all the classes (objects). It is inherited by the two other files.

2) Sound_functions.c is the first implementation of the previous file. It contains all the functions defined in the classes of the interface (except the inline functions).

3) Sound_main.c is the main program. It consists of declaring objects and calling their member-functions.

This organisation in separated files makes the program easier to understand and modify for later programmers. The permanent regard to philosophy of object oriented programming is a contributing factor to efficiency and portability.

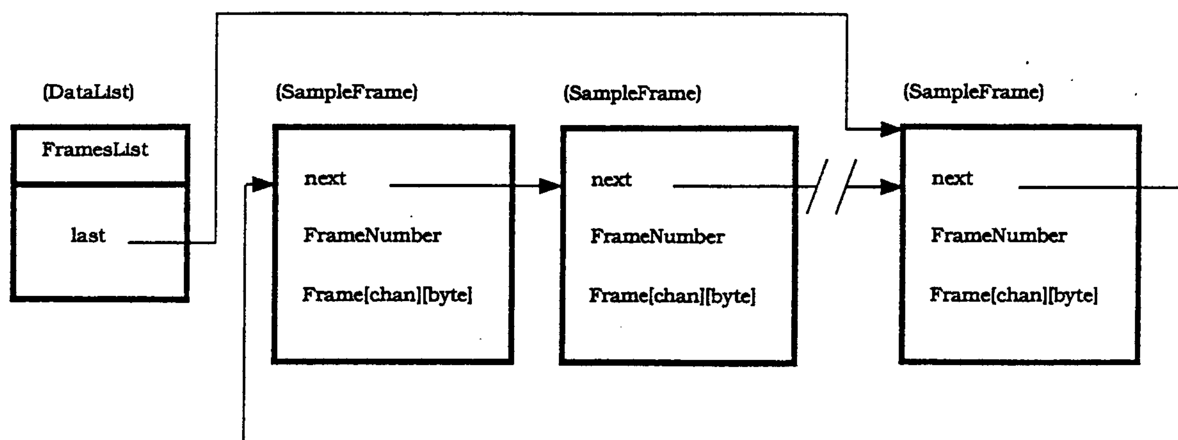
4 Object data dictionary

The Fig 2 shows an example of an AIFC sound file. We can see that an AIFC file is devided into four separated parts, called "chunks". Each chunk has its specific fields, representing the characteristics of the file.

Chunk	Field	Value
FORM AIFC file	ckID	F O R M
	ckSize	99690
	formType	A I F C
Format Version Chunk	ckID	F V E R
	ckSize	4
	timestamp	2726318400
Common Chunk	ckID	C O M M
	ckSize	38
	numChannels	1
	numSampleFrames	99611
	sampleSize	8
	sampleRate	22254.54
	compressionType	N O N E
	compressionName	14 n o t c o m p r e s s e d 0
Sound Data Chunk	ckID	S S N D
	ckSize	99619
	offset	0
	blockSize	0
	sound data	sample frames … 99611th
	padbyte	0

Fig 1 A sample AIFC-file
("Inside Macintosh",volume VI)

First we could define as many objects as there are chunks, that is to say 4 objects (**Form, FormatVersion, Common, SoundData**), according to the AIFC format of the sound files. Each one of these objects has member functions to get the attributes and the contents of these chunks and to display their data. It is convenient to store the data read from them into predefined structures. For this purpose, we defined a corresponding class for every object we mentioned above, which brings 4 more objects (**FormStruct, FormatVersionStruct, CommonStruct, SoundDataStruct**). These ones represent the actual structures that will contain the data of the AIFC files in specific corresponding fields. The size of these structures, returned for every chunk in the "friend" corresponding object by a function called every time "retSize()", is crucial because it tells the program how many bytes to read and to store into every member field. If there is a mistake somwhere (for example, a member variable is declared short instead of long, that is to say on 4 instead on 8 bytes), all the following values will be wrong.
Finally, for the sound data themselves, which of course represent the most important part of the sound files, we defined 2 other objects (**SampleFrame, DataList**). "DataList" is used to create a circular linked list of sample frames, and provides member functions to append new frames at the "end" of the list, display all the values of the frames (these values are segmented into channels and bytes) and clear (deallocate) the complete list. Since the number of frames is not known before reading the file, we had to create a linked list in order to be able to use dynamic allocation, which is necessary for such long files. Making this list circular is very practical. As we can see in Fig 1, "last->next" is always a pointer to the first frame, so that we can easily access both the last frame (with "last") to append a new frame and the first frame (with "last->next") to read the list from the beginning. The last object, "SampleFrame", is defining the frames. The frames are filled by the members of "DataList", just as the above structure-objects are filled by their corresponding objects. We can see the complete structure of the list in Fig 2.



Notes : - last->next = 1st frame (always)
- example ->
FrameNumber = 735 and
Frame[2][3] means :
2nd byte of the 3rd channel of
the 735th frame

Fig 2 Circular linked list of the sample frames

In summary, the following list gives the objects that were identified and their different fields :

1) **FormStruct** : ckID, ckSize, formType

2) **Form**: Size/Form(), retSize(), fretrieve(), print()

3) **FormatVersionStruct** : ckID, ckSize, timestamp

4) **FormatVersion**: Size/FormatVersion(), retSize(), fretrieve(), print()

5) **CommonStruct**: ckID, ckSize, numChannels, nmSampleFrames, sampleSize,sampleRate,compressionType, compressionName

6) **Common**: Size/Common(), retSize(),fretrieve(),getnumSampleFrames(), getnumChannels(), getsampleSize(), getsampleRate(),print()

7) **SoundDataStruct**: ckID, ckSize, offset, blockSize

8) **SoundData**: Size/SoundData(), retSize(),fretrieve(), getNumSoundData(), getSoundData(), print()

9) **SampleFrame**: FrameNumber, next, Frame[channel][byte], SampleFrame(), fillFrame()

10) **DataList** : last/DataList(), ~DataList(), append(), clear(), print()

5 Description of the defined functions

Here is a short presentation of the member functions of the objects described in the previous chapter :

5.1 retSize()

This function is a member of the objects Form, FormatVersion, Common and SoundData For every one of these objects it takes as an argument the corresponding structure-object (FormStruct, FormatVersionStruct, CommonStruct or SoundDataStruct). It stores in the field "Size" of the object they belong to the size of the corresponding structure-object.

5.2 fretrieve()

This function is also a member of the objects Form, FormatVersion, Common and SoundData. It opens the target file, sets the file pointer to the appropriate position, reads from the file a number of bytes equal to the size of the corresponding structure-object (FormStruct, FormatVersionStruct, CommonStruct or SoundDataStruct) and stores the data, field after field, in these structure-objects.At the end of this operation, All the field of the structure-objects contain the right value corresponding to the AIFC format. Then the file pointer is set to the beginning of the next chunk, and the same procedure can be followed in the scope of another object. An interesting thing to note here is that the definition of the strucure-objects FormStruct, FormatVersionStruct, CommonStruct and SoundDataStruct permits, using the C standard function "fread()" and giving the size of these objects as one of the arguments, to read a complete chunk in one single instruction and to automatically operate the segmentation into its different fields.

5.3 print()

This function is again a member of the objects Form, FormatVersion, Common and SoundData. It prints the contents of the strucure-objects FormStruct, FormatVersionStruct, CommonStruct and SoundDataStruct, field after field.

5.4 Common::getnumChannels()

This is an inline function that returns the (private) member numChannels by the means of an indirection (return CommonStrucureName->numChannel). This way of defining functions is typical for object oriented programming : the variables are protected (declared as private), that is to say they can only be manipulated by functions of the same class.

5.5 Common::getnumSampleFrames()

This function reads the number of sample frames from the AIFC file and stores it in 4 bytes (in a "long"). The actual value is read as an array of "unsigned char" (which is an integer type having the same size as a char, that is to say 8 bits or 1 byte). Then, step by step, every one of the four "unsigned char" values is stored in a temporary "long" variable, shifted left to the appropriate position (byte by byte) in this variable and finally appended to the "long" variable that will contain the final result.

5.6 Common::getsampleSize()

This function is similar to getnumChannels(), except that it returns the sample size instead of the number of channels.

5.7 Common::getsampleRate()

This function has to perform a conversion from the Motorola to the AT&T processor formats for floating point numbers. A complete example of this conversion can be found in annex 2. For our purpose, the sample rate is always an integer value even if it has the "double" format. So we just store the result in 4 bytes (in a "long") using an array like in the previous function, assuming that this value never has a point and skipped the 4 last bytes occupied for this field in the AIFC files. If there is a necessity for later programmers to read files with point numbers, it will be easy to modify the function according to the explanations given in the annex. The object the function belongs to doesn't have to be modified. The only thing to do is to use the last four elements of the read array that were left aside.

6 Access to the sound data

Two different ways of accessing the sound data are provided by the program :

1) Sequential access The member-function "SoundData::getSoundData"reads the input file in an array of "unsigned char" (1 byte) byte after byte untill "numSampleFrames" frames have been read. So every element of the array represents a sound point or byte. But this array doesn't make any segmentation of its points. All of them are situated on a same level and, for example, there is no way of getting the 2nd point of the 1st channel in the 735th frame. To comply with this exigency we offer another possibility : the structured access.

2) Structured access To be able to get any sound data by specifying the number of the channel and of the point in a selected frame, we introduced the circular linked list described in chapter 4. After getting the sequential array of bytes mentioned above, we create a loop of "numSampleFrames" iterations, still in ."SoundData::getSoundData". At every iteration we read from the sequential array a number of bytes equal to the product (numChannels * sampleSize(in bytes)), which gives the size of the frames in bytes, store these bytes in a temporary variable and call "DataList::append()" that will allocate memory for a new frame, append this frame to the list and, by calling "SampleFrame::fillFrame()" fill the fields "FrameNumber" and "Frame[channel][byte]" of the corresponding frame. An example of this structured access can be found in Fig 1.

7 Conclusions and recommendations

The previous chapters showed that the proposed program complies with the exigencies of the formal specifications (chapter 2). Now we make some observations and suggestions that should be useful for further developments.

- Unordered occurrences of the "chunks"

According to annex 1, in some cases, the chunks may not be ordered in the standard hierarchy that we reproduced in our program. To remedy this problem, we provide another short program, called "find.c" which fills the role of finding a string in any file and returning its position (in bytes) from the beginning. We propose a simple procedure using this program :

1) Introduce a loop of "number of chunks" (there may be more than 4) iterations. Every time, call the function "find.c" with a new chunk name and store the result in an array, let us call it "ChunkPosition".
2) Use a sort algorithm to order this array in the chronological order of appearances of the different chunks.
3) Write another loop and, at every iteration, with the help of a "switch", select the appropriate definitions and operations on the chunks as they are presented now.

- Creation of the output compressed file

Of course, the sound files will have to be compressed after restoring the data with the help of our program. So, we will have to produce the output compressed file that will really be used to produce sounds. Using the facilities of the C++ programming language, it will be easy to specify the name of the output to be created as a third argument (after the name of the executable file and the name of the target file) in the command line. The implementation "Sound_functions.c" contains also a set of error messages for output files that will be easily used provided that the new functions will be defined in objects having the appropriate form.

- Decoding of the sound points

In the AIFC format (see annex 1), if the compression type is "NONE", then each sample "point " (or byte) in a sample frame is a linear, two's complement value and is stored in an integral number of continuous bytes. Our decoding interface

restores these bytes in a convenient way. But they are not yet converted. To get their real integer value, we can follow the following steps :

1) put the first bit (sign bit) to 0 in order to get the absolute value
2) invert all the other bits with the help of bitwise operators
3) add one to the obtained value : you have the absolute value of the number

Here is an example for a value stored in 4 bits :

1) 2nd's complement value : 1101 -> 0101
2) 0101 -> 0010
3) 0010 -> 0011 => absolute value = # 3 d

The realisation of the program was facilitated to a large extent thanks to the use of the books given in the references.

The program, in its present form, provides a solid and flexible platform for applying compression algorithms. It is destined for further extensions according to the needs that will appear in the compressing operations.

A References

[1] B. STROUSTRUP, *The C++ programming language,* Bell Telephone Laboratories, Addison-Wesley, 1986

[2] Mark Williams Company, *ANSI C, A Lexical Guide,* Chicago, 1988

[3] H. NUSSBAUMER, *Informatique industrielle L, Representation et traitement de l'information,* Presses Polytechniques Romandes, Lausanne,1986

B Annexs

1 the Macintosh AIFC format : example of a sample AIFF-C file

2 Floating point numbers : example of conversion from the quadruple to the double precision representation

Annexes

Annexe 1

Chunk	Field	Value
FORM AIFC file	ckID	F O R M
	ckSize	99690
	formType	A I F C
Format Version Chunk	ckID	F V E R
	ckSize	4
	timestamp	2726318400
Common Chunk	ckID	C O M M
	ckSize	38
	numChannels	1
	numSampleFrames	99611
	sampleSize	8
	sampleRate	22254.54
	compressionType	N O N E
	compressionName	14 n o t c o m p r e s s e d 0
Sound Data Chunk	ckID	S S N D
	ckSize	99619
	offset	0
	blockSize	0
	sound data	sample frames … 99611th
	padbyte	0

Fig 1 A sample AIFC-file
("Inside Macintosh",volume VI)

Annexe 2

Floating point numbers : example of conversion from the quadruple to the double precision representation

Quadruple precision (AIFC files)

#48'000 d = #40'0E | BB'80 h
= #0100'0000'0000'1110 |
1011'1011'1000'0000 b

sign : S = 0 => positive number
exponent : E_4 = #_100'0000'0000'1110 b = #40'0E h
mantissa : M_4 = #1011'1011'1000'0000 b = #BB'80 h
(1st significant bit kept)
biais : R_4 = 16'383 (constant for this precision)

Explanation :
#48'000 d = #1.011'1011'1000'0000 * 2^{15} b (1)
(15 = shift value to restore #1011'1011'1000'0000 b)
Actual value : $A_4 = (-1)^S * M_4 * 2^{(E4 - R4)}$ (2)
48'000>0 => S = 0
Comparing (1) and (2), we find :
M_4 = #1011'1011'1000'0000 b = #BB'80 h (S=0) and
E_4-R_4 = 15 => E_4 = 15 + 16'383 = #16'398 d = #400E h
So, we can reconstitute the code :
#48'000 d = #40'0E | BB'80 h (quadruple precision)

Double precision (AT&T format)

#48'000 d = #40'07'70'00 h
= #0100'0000'1110'0111 |
0111'0000'0000'0000 b

sign : S = 0 => positive number
exponent : E_2 = #_100'0000'0000 b = #40'0 h
mantissa : M_2 = #0111 | 0111'0000'0000 b = #7'70'00 h
(1st significant bit dropped)
biais : R_2 = 1'023 (constant for this precision)

Explanation :
#48'000 d = #1.011'1 011'1 000'0 000- * 2^{15} b (3)
(15 = shift value to restore #1011'1011'1000'0000 b)
Actual value : $A_4 = (-1)^S * (1,M_4) * 2^{(E2 - R2)}$ (4)
48'000>0 => S = 0
Comparing (3) and (4), we find :
M_2 = #0111'0111'0000'0000 b = #7'77'00 h (S=0) and
E_2-R_2 = 15 => E_2 = 15 + 1'023 = #1'038 d = #40'E h
So, we can reconstitute the code :

#48'000 d = #40'07 | 70'00 h (double precision)

Number with point (case of double precision)

Let us, for example, find the value of the number 48'000.35 in the double precision format on 4 bytes.

#48'000.35 d = #1 011'1 011'1 000'0 000.0 1011 b
(#0.35 d = $0*2^{-1}+1*2^{-2}+0*2^{-3}+1*2^{-4}+1*2^{-5}+$... b = 0.3438...)
So, we find the following code :

#48'000.35 d = #40'E7'70'0B h (double precision)

Conversion

For the value "48'000", we have :
a_4 = #40'0E h and M_4 = #BB'80 h

According to Tab. 1, we can make the following conversions :

$a_2 = (a_4 - R_4) + R_2$ = (16'398 - 16'383) + 1'023 = 15 + 1'023
=> a_2 = #1'038 d = #40E h

$M_2 = M_4$ left shifted by one (omission of the 1st significant bit) and completed with zeros
=> M_4 = #487'424 d = #7'70'00 h

Now we append a_2 and M_2 according to the format of Fig 3. We obtain the double precision representation :

#48'000 d = #40'07 | 70'00 h (double precision)

as a result of the conversion.

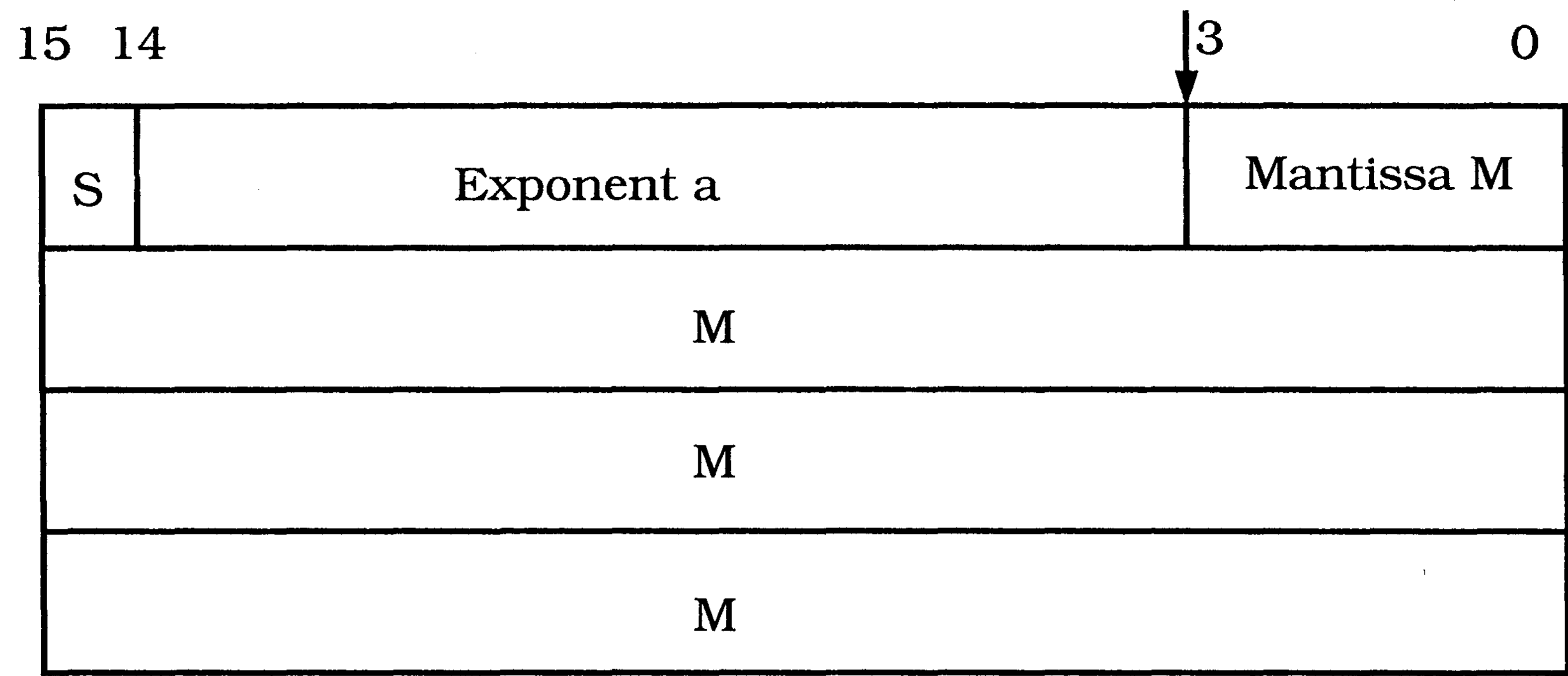


Fig 3 IEEE's floating point double representation

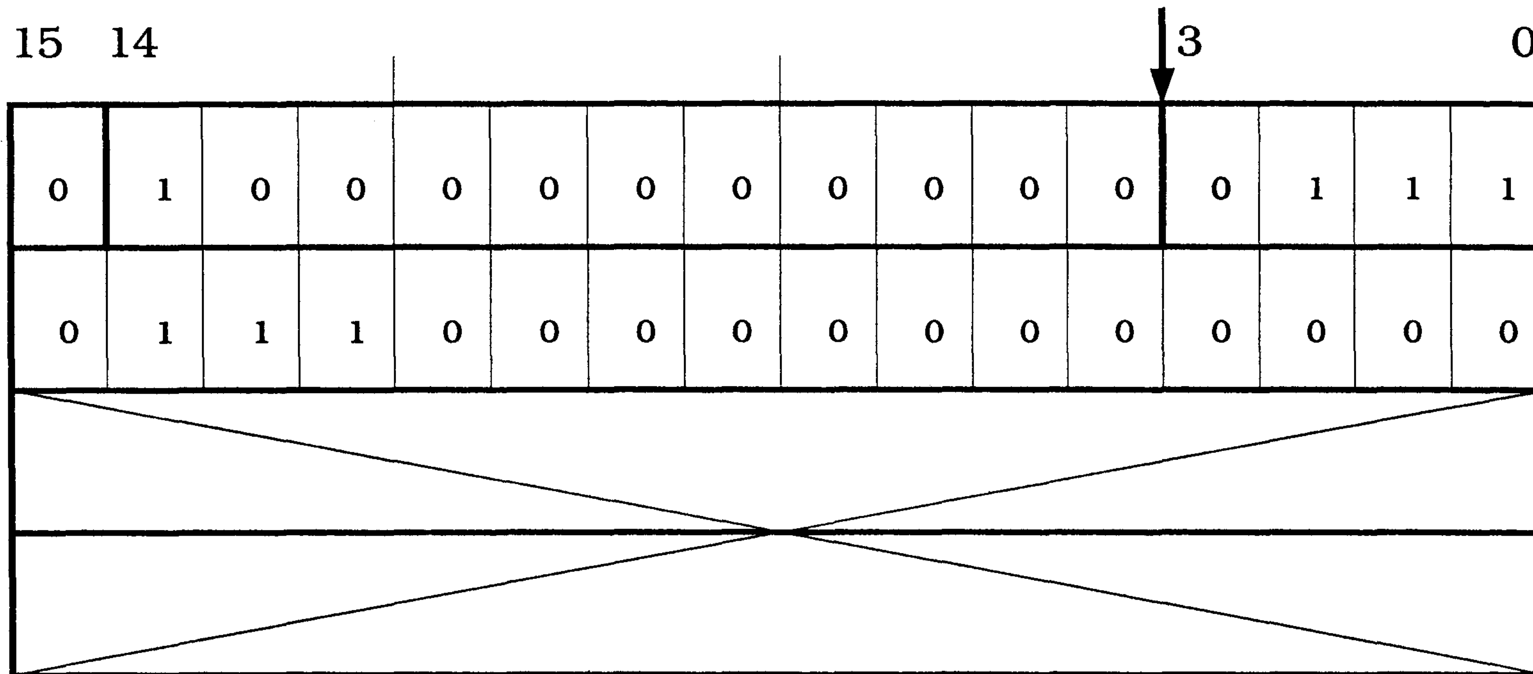


Fig 4 Representation of the number 48'000 in our program (the result is stored in along -> 4 bytes)

	DOUBLE PRECISION	QUADRUPLE PRECISION
DIMENSIONS OF THE FIELDS (in bits)		
sign S	1	1
exponent a	11	15
integer part H	/	1
fractionnary part M	52	111
total dimension	64+(1)	128
SIGN OF THE NUMBER	+ -> S = 0 - -> S = 1	+ -> S = 0 - -> S = 1
BIAIS OF THE EXPOENT R	1'023	16'383
MAXIMUM VALUE OF THE EXPONENT	2'047	32'767

Tab 1 Main characteristics of the two basics IEEE's floating point numbers format