

ClearPET Project

List Mode Format Implementation
Version 1.3.2

CONFIDENTIEL

Magalie KRIEGUER¹
Luc SIMON²
Christian MOREL²
April 15, 2002
¹*IIHE/VUB*
²*IPHE/UNIL*

Introduction

This document describes the different steps to reach the specifications that we have chosen in the Software Design report version 2.3 for the CCC ClearPET Project. It presents the different programming choices that we have made. In particular, we are going to describe how the List Mode Format is built from data given by the DAQ or the Geant4 simulation.

1 General Design

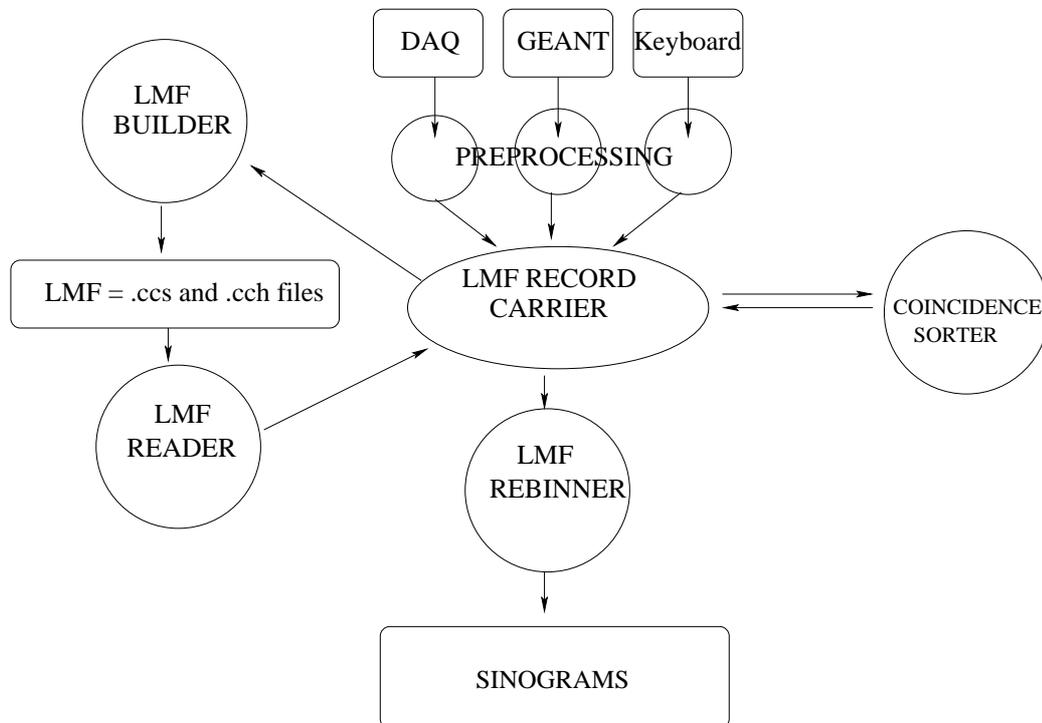


Figure 1: LMF Implementation : General Design

In order to have a workspace that is able to contain and transport all the information needed to write files in LMF, to search for coincidences, and to build sinograms, we have defined a LMF record carrier (cf figure 1) as a group of 7 C-like structures. They are described in section 3, and are used in each step of the sinogram building that we have foreseen. The LMF record carrier acts as :

- input of the LMF builder module that creates the .ccs and .cch files (cf section 2)
- input and output of the coincidence sorter module that associates events which are in coincidence
- input of the LMF rebinner module that builds the sinograms
- output of the LMF reader module that reads the .ccs and .cch files

Consequently, a preprocessing system that will be able to fill in the record carrier for each different sources of data (DAQ and GEANT4) will have to be implemented.

2 List Mode Format

The List Mode Format (LMF) contains all information for one acquisition : the records themselves, but also the data acquisition or simulation parameters, basic information concerning the animal, and much more.... The LMF is composed of 2 files : the LMF ASCII header file and the LMF binary scan file, with extensions .cch and .ccs, respectively.

2.1 The LMF ASCII header file (.cch)

The file with extension .cch is an ASCII file that contains a list of floating point and string information about the scan, the acquisition, and the animal. This file contains lines formatted as *name of the field : contents of the field*, such as for example :

```
scanner identification : ClearPET GLS  
subject weight : 150 g  
scan start time : 12:53:07  
scan date : Sep 11 2002
```

2.2 The LMF binary scan file (.ccs)

The file with extension .ccs is a binary file that contains event wise integer information encoded in records, and parameters necessary to decode these records. The encoding of these data is stored in the LMF encoding header which is described in section 3 of *Software Design version 2.3*. We can summarize it in the example given below (16 bit words are given in hexadecimal) :

e601 : encoding rule for detector ID corresponding to :

Exemple : 1110 0110 0000 0001 \rightarrow sssM Mmmc cccc cccl.

That means : 3 bits reserved for rings and sectors within rings, hereafter referred to as rsectors, 2 for modules, 2 for submodules, 8 for crystals and 1 for layers.

f67f : scanner description encoded using the above example rule. Here 8 rsectors, 3 modules/sectors, 4 submodules/module, 64 crystals/submodule, and 2 layers/crystal.

f00f : tangential scanner description. Here 8 rsectors/ring, 3 rows of modules/sector, 1 row of submodule/module, 8 rows of crystals/submodule, and 2 layers/crystal.

060e : axial scanner description. Here 1 ring, 1 column of modules/ring, 4 columns of submodules/module, 8 columns of crystals/module (the layer bits are always set to 0).

0002 : number of different record types. Here 2 types.

0e30 : encoding pattern for event records using the rule described in section 3.1.3 of *Software Design version 2.3* :

TTTT cdEn NNgb sRRR. Following this rule, event records consist in coincidences with detector IDs and energies, and azimuthal and axial positions of the gantry. The tag of event records is **0**.

8d60 : encoding pattern for count rate records using the rule described in subsection 3.1.4 of *Software Design version 2.3* : *TTTT sSSc FrbR RRRR*. Following this rule, count rate records consist in singles count rates for all sectors, total coincidence count rate, and axial and angular speed of the bed/gantry mechanics. The tag of count rate records is **8**.

Then, the various records are themselves encoded as described in section 3.1.3 (event record) and 3.1.4 (count rate record) of *Software Design version 2.3*.

3 LMF record carrier

The LMF record carrier is a “cluster” of 7 structures which are detailed below. The first structure contains information about the LMF header file (.cch), and the 6 other ones contain information related to the LMF binary scan file (.ccs).

3.1 LMF_cch structure

The DAQ provides all the relevant information about the scan, such as the scan file name, the scan date, the tracer identification, the injected dose, etc. (*cf. page 4 of Software Design Version 2.3/January 2002*). The LMF builder uses parameters contained in the LMF_cch structure of the LMF record carrier to create the LMF ASCII header file (.cch).

```
struct LMF_cch {
    char field[charNum]; /*string field description*/
    char data[charNum]; /*string field content*/
    char unit[charNum];
    char def_unit[charNum]; /*default unit used by the
LMF rebinner*/
    VALUE value; /*numerical value*/
    VALUE def_unit_value; /*numerical value
converted in default unit value*/
};

typedef union {
    char vChar[charNum]; /*value is a string*/
    float vNum; /*value is a floating type*/
    struct tm tps_date; /*value is a date or a time*/
} VALUE;

LMF_cch scanHeader;
```

We have defined as symbolic constants a list of default units. These units are used to convert the numerical values of the LMF header. Units are identified by their string representations as listed in the table below, and concatenated with a standard prefix a, f, p, n, mu, m, c, d, da, h, k, M, G, T, P which defines the order of magnitude :

	Possible Unit	Default Unit
Energy	eV, J, Wh, cal, erg	keV
Distance	m, in, ft	mm
Surface	[Distance] ²	mm ²
Volume	[Distance] ³	mm ³
Time	h, min, s	s
Activity	Bq, Ci	MBq
Speed	[Distance]/[Time]	mm/s
Angle	degree, rad, grad	rad
Rotation Speed	[Angle]/[Time], rph, rpm, rps	rad/s
Weight	g, oz, lb	g
Temperature	C, F, K	C
Electric field	V	V
Magnetic field	gauss, T	T
Pression	Pa, atm, bar, mmHg	hPa

For example, if the user sets an injected dose of 10 mCi, the LMF record carrier is loaded with :

- scanHeader.field = "injected dose"
- scanHeader.data = "10 mCi"
- scanHeader.unit = "mCi"
- scanHeader.def_unit = "MBq"
- scanHeader.value = 10.0
- scanHeader.def_unit_value = 360.0

3.2 LMF_ccs_encodingHeader structure

This structure contains 3 other structures that are common to an acquisition. It means that these structures contain encoding ID information, and basic topological parameters of the PET scanner.

```
struct LMF_ccs_encodingHeader
{
struct LMF_ccs_scanEncodingID scanEncodingID;
/*sssMMMMmmccccl=1110000111000001*/
struct LMF_ccs_scannerTopology scannerTopology;
/*scanner Design*/
struct LMF_ccs_scanContent scanContent;
/*definition of the records*/
};
```

3.2.1 LMF_ccs_scannerTopology structure

We find here low level information about the scanner topology :

- number of rings
- number of sectors/ring
- number of modules/sector
- number of submodules/module
- number of crystals/submodule
- number of layers/crystal

In all the software, we consider the scanner as its flat development, corresponding to the result of cutting the rings, and unfolding them as shown in figure 2.

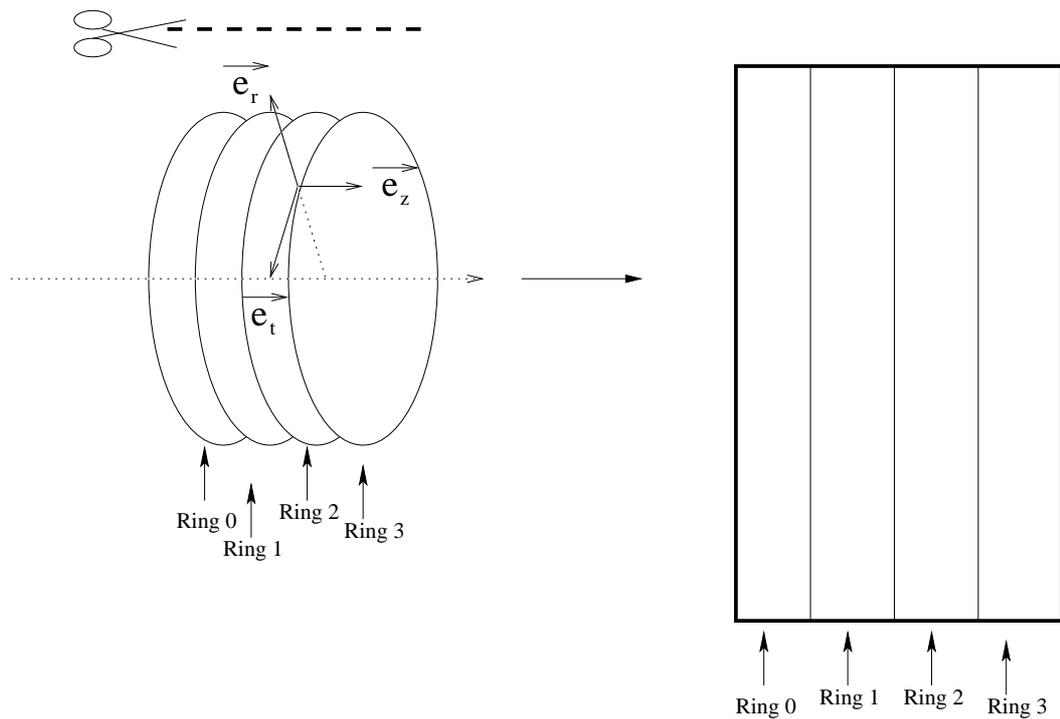


Figure 2: Numbering of rings. Example of view of the unfolded scanner

Using the representation of rsectors, we will distinguish between the number of sectors (i.e rows of rsectors tangentially) and the number of rings (i.e columns of rsectors axially).

Then the numbering of the modules, submodules, or crystals uses the same paradigm. As an example, figures 3 and 4 describe a topology of a “1 ring scanner” with 8 sectors that we will call scanner A. Each sector has 3 rows of modules tangentially, and only 1 column of modules axially. Each module is divided in 4 columns of submodules (and 1 row tangentially). Finally each submodule is divided in a matrix of 8 rows by 8 columns of crystals. The numbering of crystals is shown on figure 3.

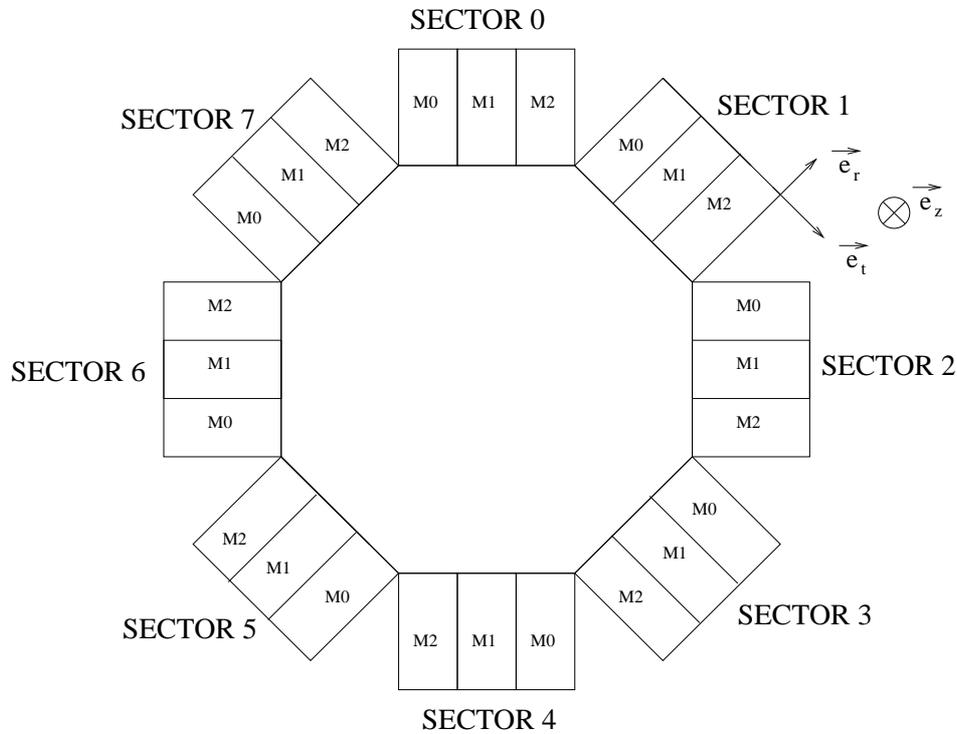


Figure 3: Numbering of the sectors and modules. Axial view of scanner A

As you can see the numbering is always (for rings, sectors in a ring, modules in a sector, submodules in a module and crystals in a submodule) starting tangentially, and grows along the axis directions specified in figures, 3, and 4.

For the layers, we will number 0 the internal layer (close to the axis) and 1 the external one.

Finally, the structure that contains the number of rings, sectors, modules, submodules, crystals and layers is :

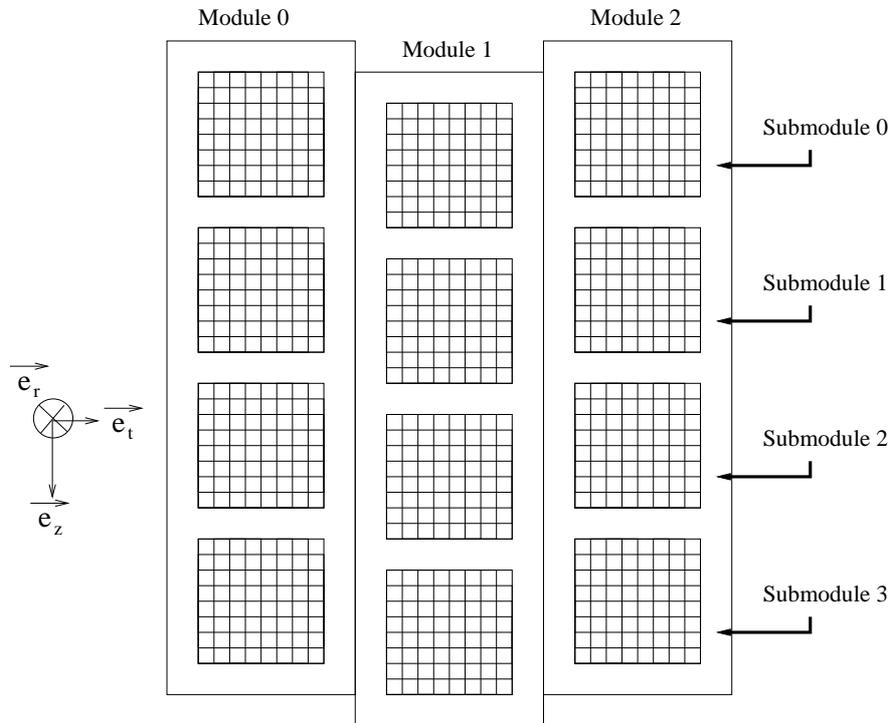


Figure 4: Numbering of the modules and submodules. Tangential view of a sector of scanner A

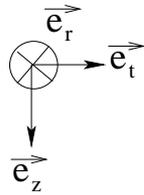
```

struct LMF_ccs_scannerTopology {

  unsigned char numberOfRings ;
  /*rings*/
  unsigned char numberOfSectors ;
  /*sectors*/
  unsigned char totalNumberOfRsectors ;
  /*rings * sectors*/

  /*----- FOR EACH SECTOR : -----*/
  unsigned char axialNumberOfModules ;
  /*modules axially*/
  unsigned char tangentialNumberOfModules ;
  /*modules tangentially*/
  unsigned char totalNumberOfModules ;
  /*modules per sector*/

```



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Figure 5: Numbering of the crystals in a submodule for scanner A

```

/*----- FOR EACH MODULE : -----*/
unsigned char axialNumberOfSubmodules ;
/*submodules axially*/
unsigned char tangentialNumberOfSubmodules ;
/*submodules tangentially*/
unsigned char totalNumberOfSubmodules;
/*submodules per module*/

/*----- FOR EACH SUBMODULE : -----*/
unsigned char axialNumberOfCrystals ;
/*crystals axially*/
unsigned char tangentialNumberOfCrystals ;
/*crystals tangentially*/
unsigned char totalNumberOfCrystals ;
/*crystals per submodule*/

/*----- FOR EACH CRYSTAL : -----*/

unsigned char axialNumberOfLayers ;
/*layers axially (always 1)*/
unsigned char tangentialNumberOfLayers ;
/*layers tangentially*/
unsigned char totalNumberOfLayers ;
/*layers per crystal*/
};

```

3.2.2 LMF_ccs_scanEncodingID structure

This subsection describes how the encoding format of crystal and DOI is implemented. In the example of section 2.2, we have previewed the encoding rule *sssM MMMm mmcc cccl*. This means :

- 3 bits of the ID reserved for the rsectors (8 rsectors maximum)
- 4 bits of the ID reserved for modules (16 modules maximum)
- 3 bits of the ID reserved for submodules (8 submodules maximum)
- 5 bits of the ID reserved for crystals (32 crystals maximum)
- 1 bit of the ID reserved for layers (2 layers maximum)

But the current implementation will also work if we change, for example, this format by *sssM Mmmc cccc cccl*, which means :

- 3 bits of the ID reserved for the rsectors (8 rsectors maximum)
- 2 bits of the ID reserved for modules (4 modules maximum)
- 2 bits of the ID reserved for submodules (4 submodules maximum)
- 8 bits of the ID reserved for crystals (256 crystals maximum)
- 1 bits of the ID reserved for layers (2 layers maximum)

This is the aim of the scan_encodingID structure :

```
struct LMF_ccs_scanEncodingID { /*sssM MMMm mmcc cccl*/
unsigned char bitForRsectors ;
/*number of bits for rings/sectors in ID*/
unsigned short maximumRsectors ;
/*2 ** bitForRsectors*/
unsigned char bitForModules ;
/*number of bits for modules in ID*/
unsigned short maximumModules ;
/* 2**bitForModules*/
unsigned char bitForSubmodules ;
/*number of bits for submodules in ID*/
unsigned short maximumSubmodules ;
/*2**bitForSubmodules*/
unsigned char bitForCrystals ;
/*number of bits for crystals in ID*/
unsigned short maximumCrystals ;
/*2**bitForCrystals*/
```

```
unsigned char bitForLayers ;  
/*number of bits for layers in ID*/  
unsigned short maximumLayers ;  
/*2**bitForLayers*/  
};
```

3.2.3 LMF_ccs_scanContent structure

This structure shows the types of records stored (or not) in the acquisition, and the tag of these records. This tag is used in the binary scan file (.ccs) to recognize which type of records is read. Presently, there are only 2 types of records (event record and count rate record), but it will be possible to define other types of records.

```
struct LMF_ccs_scanContent {  
  
    unsigned char nRecord ;  
    /*number of different records*/  
  
    /*first record : event record*/  
    unsigned char eventRecordBool ;  
    /*event recorded if 1*/  
    unsigned char eventRecordTag ;  
    /*event tag=0(1st bit of encoding event)*/  
  
    /*second record : countrate record*/  
    unsigned char countRateRecordBool ;  
    /*countrate recorded if 1*/  
    unsigned char countRateRecordTag ;  
    /*countrate tag = 1000 (4 bits)*/  
  
    /*+ eventually other records */  
};
```

3.3 LMF_ccs_currentContent

This structure contains just one field. It indicates what type of records is currently loaded in the LMF record carrier.

```
struct LMF_ccs_currentContent  
{  
    unsigned char typeOfCarrier;  
};
```

The field typeOfCarrier is equal to the tag of the record type (0 for event record and 8 for count rate record)

3.4 LMF_ccs_eventHeader structure

This structure contains all the parameters of the event record acquisition. It is more or less a list of booleans, where we can find the answer to questions like :

- Do we want to store the detector's ID in this acquisition ?
- Do we want to store energy ?
- ...

Finally, all the information needed to build the event encoding header described in *Software design version 2.3* is in this structure.

Note that the numbering of the neighbours of a crystal is described in the figure 6.

0	<pre> 1 4 0 2 3 </pre>	<pre> 5 1 6 4 0 2 8 3 7 </pre>	<pre> 11 12 13 10 5 1 6 14 9 4 0 2 15 20 8 3 7 16 19 18 17 </pre>	Numbering
0	1	2	3	Neighbourhood order
0	4	8	20	Number of neighbours

Figure 6: Numbering of a crystal neighbours.

```

struct LMF_ccs_eventHeader {
    unsigned char coincidenceBool;
    /*coincidence if 1, singles if 0*/
    unsigned char detectorIDBool;
    /*detector ID recorded if 1*/
    unsigned char energyBool;
    /*energy recorded if 1*/
    unsigned char neighbourBool;
    /*energy of neighbours recorded if 1*/
    unsigned char neighbourhoodOrder;
    /*0, 1, 2 , or 3 (cf fig. 1)*/
    unsigned char numberOfNeighbours;
    /*Number of neighbours*/

```

```
    unsigned char gantryAxialPosBool;
/*gantry's axial position*/
    unsigned char gantryAngularPosBool;
/*gantry's angular position*/
    unsigned char sourcePosBool;
/*source's position*/
};
```

3.5 LMF_ccs_eventRecord structure

This structure must accept any event record. It must contain all the information needed to build an event record for the binary scan file (.ccs). Here is the structure we have figured out. You can notice that the energy, and the crystal IDs will be stored in an array dynamically allocated from the pointers **energy* and **crystalID*. The maximum dimension of these 2 arrays is 2 x 21, to store information on the central crystal and its 20 neighbours for the first and the second annihilation photons.

```
struct LMF_ccs_eventRecord {
    unsigned char timeStamp[8] ;
/* time stamp on 63 bits but maybe less...*/
    unsigned char timeOfFlight ;
/* time of flight*/
    unsigned short *crystalIDs;
/*crystal's ID (1st & 2nd and neighbours)*/
    unsigned char *energy;
/*energy in each crystal*/
    unsigned short gantryAxialPos ;
/*gantry's axial position*/
    unsigned short gantryAngularPos;
/*gantry's angular position*/
    unsigned short sourceAngularPos ;
/*external source's angular position*/
    unsigned short sourceAxialPos ;
/*external source's axial position*/
};
```

Warning : the time stamp in the LMF binary scan file is stored on 63 bits for singles events (absolute time in units of about 1 ps), and on only 23 bits for coincidence events (relative time to the previous event record in units of about 1 ms).

3.6 LMF_ccs_countRateHeader structure

It contains all the information needed to build the count rate encoding header (exactly as the eventHeader structure for the event encoding header).

```
struct LMF_ccs_countRateHeader {
    unsigned char singleRateBool ;
    /*singles countrate recorded if =1*/
    unsigned char singleRatePart ;
    /*rsector (1), module(2),
    submodule(3) or total (0)*/
    unsigned char totalCoincidenceBool ;
    /*total coincidence recorded if =1*/
    unsigned char totalRandomBool ;
    /*total random rate recorded if =1*/
    unsigned char angularSpeedBool ;
    /*angular speed recorded if =1*/
    unsigned char axialSpeedBool ;
    /*axial speed recorded if =1*/
};
```

3.7 LMF_ccs_countRateRecord structure

This structure must accept any information from count rate record. We can notice that the singles rate per ring, per sector, or per module will be stored in an array dynamically allocated from the corresponding pointers of this structure.

```
struct LMF_ccs_countRateRecord {
    unsigned char timeStamp[4];
    /*time stamp*/
    unsigned short totalSingleRate[2] ;
    /*total single rate*/
    unsigned short *pRsecorRate;
    /*rsector's rate pointer*/
    unsigned short *pModuleRate;
    /*module's rate pointer*/
    unsigned short *pSubmoduleRate;
    /*submodule's rate pointer*/
    unsigned short coincidenceRate ;
    /*coincidence rate*/
    unsigned short randomRate ;
    /*random rate*/
    unsigned char angularSpeed ;
    /*gantry's angular speed*/
```

```
    unsigned char axialSpeed ;  
/*gantry's axial gantry*/  
};
```

4 Other Modules

4.1 LMF reader

This module is able to read a LMF ASCII header file (.cch) and/or LMF binary scan file (.ccs), and to load the LMF record carrier, record by record, using a buffer dynamically allocated to accept the size of the records.

4.2 LMF builder

This module must write the LMF files (.cch and .ccs) from the content of the LMF record carrier. It is buffering each record before writing it to the file.

4.3 Coincidence sorter

This module should read different event record carriers and update those after having associated coincidences between event records of singles.

This is done by using the time stamp of the singles event records. Consequently, this module should buffer enough record carriers to be able to associate coincidences. Its output is a record carrier that hosts a coincidence event.

4.4 LMF rebinner

This module should finally build the sinograms after the LMF files have been read by the LMF reader. To build the sinograms, the record carriers have to be histogrammed. Once the sinogram is built, we will use the STIR library to reconstruct images. The LMF rebinner will save the sinograms in an interfile output format compatible with the STIR library (<http://stir.irsl.org>).

Contents

1	General Design	1
2	List Mode Format	2
2.1	The LMF ASCII header file (.cch)	2
2.2	The LMF binary scan file (.ccs)	2
3	LMF record carrier	4
3.1	LMF_cch structure	4
3.2	LMF_ccs_encodingHeader structure	5
3.2.1	LMF_ccs_scannerTopology structure	6
3.2.2	LMF_ccs_scanEncodingID structure	10
3.2.3	LMF_ccs_scanContent struture	11
3.3	LMF_ccs_currentContent	11
3.4	LMF_ccs_eventHeader structure	12
3.5	LMF_ccs_eventRecord structure	13
3.6	LMF_ccs_countRateHeader structure	14
3.7	LMF_ccs_countRateRecord structure	14
4	Other Modules	15
4.1	LMF reader	15
4.2	LMF builder	15
4.3	Coincidence sorter	15
4.4	LMF rebinner	15