The BOSIO System for CLAS detector - draft

Memory management and Input/Output

Serguei Boiarinov, Vladimir Serov
Institute for Theoretical and Experimental Physics
Moscow, Russia

....
TJNAF
Newport News, VA

October 3, 1996

Abstract

BOSIO is a program system, written in C language, for the memory management of data banks and for the input/output of sets of data banks. The system follows by the same ideology as Volker Blobel's BOS system and creates files in FPACK format. In additional present system has the number of extensions to dealing with CODA online DAQ system and Data Distribution (DD) system. The main application is in the data handling in online system of the CLAS detector.

Contents

List of Tables

1 Introduction

BOSIO system are based on the same idea as standard BOS [1] system. Main difference: all banks are stored not in COMMON block which is global section, but can leaves in any allocated memory segments. Using this system user can work directly in DD system event buffer which has corresponding format, and also there is possibility to have access to data banks in Readout Controller for users programs which are called from readout list.

The basic operations,provided by the system, the same as for BOS system:

- create a bank,
- find a bank,
- delete (drop) a bank, and
- garbage collection.

The main new features are the following:

1. BOSIO structure can be created in any allocated memory segment.
2. Unlimited number of arrays can be used (in frame of existing memory).
3. The system is written in C language and tested on ULTRIX, HP-UX, SunOS, AIX and VXWORKS operating systems.
4. A table structure can be specified for banks.
5. New I/O package is introduced instead of using FPACK.

Backward compatibility with the FORTRAN77 version of the BOS system are described in appendix.

[1] Volker Blobel, The BOS System, II. Institut für Experimentalphysik, Universität Hamburg

2 Initialization

2.1 General rules

Arguments

Most of the functions act on an BOSIO-array specified in the first argument, which is (int *) pointer. Further arguments are either of type integer or of type (char *), with one exception: the index of a work bank, which is (int *). For (char *) arguments you can use both character array and character string.

Return codes

Error conditions, meaning that the subprogram could not execute the required operation, are indicated by returned negative value. This return code is explained in each section together with the call. For some errors, where a continuation is impossible (for example, if the bank structure is destroyed), the program will stop after printing the reason.

2.2 BOSIO array initialization

If you have any allocated memory segment you can create BOSIO array in this segment calling bosInit() function, which will return negative value if error or zero if everything O'k.

Arguments

```
jw     - pointer on the memory segment
nwords - the size of BOSIO array ( should be less then size of allocated
         memory segment )
```

Function call

```
status = bosInit(int *jw, int nwords);
```

Further calls of bosInit(jw,nwords) with the same array pointer "jw" will reset the array to the initial state, using the given size parameter.

By a call of bosInit(jw,0) you can check is array "jw" has BOSIO array already. It can be useful if your program got pointer on some buffer created by another program. If BOSIO array exist bosInit() will return zero.

3 Named banks

Named banks are the basic elements of the BOSIO system. The structure of named banks is shown in table 3. The system includes subprograms for various operations with single banks and sets of named banks including input/output.

3.1 Operations with single named banks

Operations with single named banks require as arguments the name and the number of the named bank. They are performed by functions, which start with the bosN. Immediately after the function call a test on ind != 0 should follow in order to check the success of the operation.

Arguments

```
jw   - pointer on the BOSIO array
name - pointer on the name of bank ( 4 characters )
nr   - the number of bank
ncol - the number of columns in a bank (not negative)
nrow - the number of rows in a bank (not negative)
ind  - index of a named bank
```

Function calls

```
ind = bosNcreate (int *jw, char *name, int nr, int ncol, int nrow)
```

- create a named bank with ND=ncol*nrow data words for I,F and B32 formats, ND=((ncol+1)/2)*nrow data words for B16 format and ND=((ncol+3)/4)*nrow data words for B08 format and to return the index of the new bank. If the bank exists already, the number of data words is changed to the value given in the argument. All additional words or all words in the case of a new bank are set to zero. If space is insufficient (even after a garbage collection in the work bank area) to create a new bank or to increase the number of data words, the returned value is zero and the return code JW(2) is set to 1 or 2; in this case a new bank is not created and an existing bank is left unchanged (note that in the latter case the bank will exist after return, the returned index however is zero).

```
ind = bosNlink   (int *jw, char *name, int nr)
```

- determine the index "ind" of a named bank. The return value is zero, if the value is zero, if the bank does not exist; therefore a test on ind==0 should follow. Other methods for the determination of the index of a bank are discussed later.

```
ind = bosNdrop   (int *jw, char *name, int nr)
```

- named bank is dropped. Although it may still be in the BOSIO array, its index cannot be obtained by bosNlink() and the bank will vanish at the next garbage collection.

```
ind = bosNprint  (int *jw, char *name, int nr)
```

- named bank is printed. If a format is defined for the bank name, it is used.

3.2 Indices for named banks

The standard method to determine the index of a given bank is to use the functions bosNlink. Although this function are fast (one call is equivalent to about two calls of a dummy function), there are certain applications with loops etc, where the total time spent in the function may become large. The algorithm for indices of named banks is explained below and it will be shown, how function calls can be avoided.

Name-indices

To each name appearing in a call of BOSIO system programs a name-index with values 51, 52 ... is assigned. This assignment is made at the first appearance of a name. The name-index for a given name "name" can be obtained by the statement

```
nami = mamind(int *jw, void name)
```

- "name" here can be (int) or (char *).

If banks are existing for a given name, the index of the first bank (first means lowest number) is stored in word w[nami] (here and below w = jw - 1). Thus having determined the name-index "nami" for a given name by the above statement, the statement

```
ind = w[nami]
```

directly yields without a function call the index of the first bank, or zero, if no bank of the given name exists. During the execution of a program the name-index will never change. Therefore within a subprogram the statements

```
nami = 0;
.....
if(nami == 0) nami=mamind(jw,name);
```

can be used to determine the name-index at the first call of the subprogram, in order to save time.

The word w[ind-1] in each bank contains the index of the next bank (in increasing order of bank numbers) of the same name, or zero, if no further bank exists. Thus by the statement

```
ind = w[ind-1];
if[ind == 0] ...
```

the next index can be determined without a function call.

Loops

If a loop has to be executed over all banks of a given name (in order of increasing number) one has to initialize the index "ind" before the loop to mamind(jw,name)+1 or nami+1; the following statements can then be used for the loop:

```
   ind = nami+1;       /* initialize index ''ind'' */
m: ind = w[ind-1];     /* next index */
   if(ind != 0)        /* test termination of the loop */
   {
      ...              /* ''ind'' is index for the bank */
      ...              /*     with number w[ind-2]      */
      goto m;
   }
```

Note that a loop can be executed without any function call for the determination of indices.

3.3 Bank formats

The format of a bank describes the type of data stored in the bank. All banks of the same name have to be described by the same format. This requirement of course has consequences for the design of banks. The definition of a format is essential for machine-independent input/output, it is useful for the printing of banks and if a bank is copied to a work bank. In machine-independent output, the data words are converted according to the format to a standard 32-bit representation, and in machine-independent input the data words are converted to the internal machine representation. It is recommended, not to use complicated formats (although it is possible), and to prefer the floating-point format.

The format of a bank can be either *mixed* or *bit-packed*. The type of the data words is described by a character string "fmt".

Mixed format

The format description "fmt" follows the same rules as FORTRAN-format statements, except that no length information is specified:

I	integer
F	floating-point
A	text (4 characters per word)

The character string defining the format is of the type

$$'f_1, f_2, \ldots',$$

where the f_i are format codes nI, nF, nA or $n(\ldots)$ for a group format specification (n = constant), for example:

'I,5F,2(I,3F)'

Outer parentheses of the format can be omitted. Note that, according to the FORTRAN rules, when format control reaches the last (outer) right parenthesis and there are data words left, the format starts again by the last preceding right parenthesis, including its group repeat count, if any, or, if no group specification exists, then at the first left parenthesis of the format specification.
If a bank contains only text, the use of the format '18A' is recommended, because in this case the printing routines use a format without gaps.

Bit-packed format

The 32 rightmost bits of each machine word are used, they contain either 32-bit words or two 16-bit words or four 8-bit words. The format description is:

'B32'	for 32-bit words
'B16'	for two 16-bit words
'B08'	for four 8-bit words

In machine-independent output the content (32 rightmost bits) of the machine words are transmitted without conversion. In machine-independent input, the data are transmitted to the 32 rightmost bits of each machine word. In printing named banks with a bit-packed format the content of the banks are printed unpacked.

Arguments

```
name   = name of bank
fmt    = character string for mixed format
       = 'B32' or 'B16' or 'B08' for bit-packed format
```

Subroutine call

```
status = bosNformat (int *jw, char *name, char *fmt)
```

The bank format for the given name is defined. The call is ignored, if the format has already been defined by a call. On output the format descriptions are added automatically to the records, in a input the format description are read. Thus formats are also defined by input of records. However, the first call of bosNformat overwrites a format defined from a record. If the format description in a call has a syntax error, the program will stop at the first usage of the erroneous format description.

Examples: The interpretation of several format strings is given.

```
                                          format of data words:

bosNformat(jw,'HEAD','2I,3F')               IIFFFIIFFFIIFFF... (repeat IIFFF)

bosNformat(jw,'DATA','F')                   FFFFF... (all floating-point)
```

```
bosNformat(jw,'DATA','(F)')                FFFFF... (all floating-point)

bosNformat(jw,'TRAC','2I,A,2(2I,F),(F)')   IIAIIFIIFFFFFF...

bosNformat(jw,'RAWD','B16')                16-bit packed
```

3.4 Sets of banks

For various operations (input/output, printing etc.) it is necessary to define a set of banks, which belong together. In BOSIO a set of banks is defined by the list of all names of banks, which belong to the set. This is simple, but not completely general: the case where some banks of a given name are included in the set of banks and others of the same name are not included in the set, cannot be described. If, in any operation, the list contains names of banks, which are not existing, these names are ignored.
A list of names may be an internal list or an explicit string of names, for example 'HEADTRACLIST' for the names HEAD, TRAC and LIST. Internal lists are specified by a single character L; the character can have one of the values

C E R S T and 0 (empty list)

The names of internal lists are stored in the BOSIO array. The subroutine bosLctl() can be used to edit an internal list (add names, delete names). The function bosLname() can be used to obtain the n-th name of a list.

Arguments

```
opt    = 'L='   set internal list L to ``list''; L = C E R S or T
       = 'L+'   add list ``list'' to internal list L
       = 'L-'   delete list ``list'' from internal list L
list   = explicit string of names
       = 'L' character, specifying the internal list; L = C E R S T 0
n      = index within a list
```

Subprogram calls

```
bosLctl (int *jw, char *opt, char *list)
```

The internal list specified within “opt” is either set equal to the list “list” ('L='), or the names from list “list” are added to or deleted from the internal list ('L+' or 'L-')

```
char *name;
.....
name = bosLname(int *jw, int n, char *list)
```

The name of the n-th entry in the list “list” is returned. The value blank (name = ' ') is returned, if n < 1 or n > the number of names in the list.

```
status = bosLdrop (int *jw, char *list)
```

All banks with names in the list “list” are dropped.

```
status = bosLprint (int *jw, char *list)
```

All banks with names in the list “list” are printed.

Examples: A few calls of bosLctl() are shown, together with the content of the internal list E after each call.

```
                              content of list E after call

bosLctl(jw,'E=','HEADRAWD')   HEAD RAWD

bosLctl(jw,'R=','SHOW')
```

`bosLctl(jw,'E+','R')`	`HEAD RAWD SHOW`
`bosLctl(jw,'E+','TRAC')`	`HEAD RAWD SHOW TRAC`
`bosLctl(jw,'E-','SHOW')`	`HEAD RAWD TRAC`
`bosLctl(jw,'E+','TRAC')`	`HEAD RAWD TRAC`
`name = bosLname(JW,2,'E')`	`returned function value is name='RAWD'`
`bosLctl(jw,'E=','0')`	`empty`

3.5 Garbage collection

Garbage collection in the area of named banks regains the space occupied by dropped banks. The position of remaining banks may change by this operation. Garbage collection for the area of named banks is not done in any system subprogram (with the exceptions of the general input-output program FSQER) and has to be called explicitly by the user. In typical applications this is done by the user in regular intervals, for example before a new record is read in.

```
status = bosNgarbage (int  *jw)
```

4 Work banks

Work banks are created with an individual index "id". They should be used preferentially inside a program module and *not* for the transmission of data between modules, since their properties may contradict the requirements of modularity and no input/output is foreseen. They are however useful and efficient inside a module; they do not disturb the area of named banks and allow a garbage collection with automatic update of all indices of work banks. The structure of a work bank is shown in table 5. Garbage collection in the region of the work banks is done automatically, if neccessary, with update of all individual indices of existing works banks. When a work bank is created, the index of the work bank is stored in "id", and the address of the word "id" is stored in the work bank at w[id-1]. This allows the system to update the content of indices "id" automatically during a garbage collection. This method requires, that the content of the work bank is referenced directly with the index "id", which was used during the creation of the bank. If the argument "id" is nonzero at entry to a work bank subroutine, it is checked: it should point to a work bank, and word w[id-1] should contain the address of the word "id". If the value of "id" is incorrect, the program will stop. Before a work bank is created, the index "id" must have the value zero. The value of an index should *never* be changed by the user directly. The indices "id" of all work banks should be in global sections; otherwise there are potential problems with certain optimizing compilers (index "id" not updated during a garbage collection).
Work bank indices or the work bank data part should not be an argument in a subprogram, if the subprogram itself may create or drop a work bank.

4.1 Operations with work banks

Arguments

```
id     = index of the work bank
ncol   = the number of columns in the work bank (not negative)
nrow   = the number of rows in the work bank (not negative)
```

Subprogram calls

```
int id=0;
.....
status = bosWcreate (int *jw, int *id, int ncol, int nrow);
```

A work bank with ND=ncol*nrow data words is created, the index is stored in "id". If the bank already exists (id $\neq 0$ at entry), the number of data words will be changed to the value of ND=ncol*nrow given in the argument. For a successfull operation, the normal return is taken with index "id" defined.
The alternate return is taken in case of insufficient space; if space is insufficient (even after a garbage collection in the work bank area) to create a new bank or to increase the number of data words, the return code is set to 1 or 2 and in this case a new bank is not created and an existing bank is left unchanged.

```
status = bosWdrop (int *jw, int *id)
```

The work bank at index "id" is dropped and "id" is set to zero. If id = 0 at entry, immediate return is taken.

```
status = bosWprint (int *jw, int id)
```

The work bank at index "id" is printed. As an exception the index may also be an index of a named bank.

A garbage collection in the area of work banks is done automatically, when necessary. The user never has to call the garbage collection directly, but for completeness the call is given.

```
status = bosWgarbage(int *jw)
```

Example: A work bank with 1000 words is created and used. After the use the work bank is printed and dropped.

```
int id=0;
.....
status = bosWcreate(jw, id, 1000, 1);
w[id+1]=31415;
.....
bosWprint (jw, id);
bosWdrop  (jw, id);
```

5 Input/Output using FPACK format

This chapter explains the subprograms for the input/output of sets of named banks, which are defined by lists of names. FPACK format [2] are used for all input/output operations with banks. During input/output operations a buffer is used and existing format description of banks are automatically written and therefore available in subsequent programs reading the data set.

5.1 Read and write

Arguments

```
descriptor - file descriptor
jw         - adress of the BOSIO array
list       - list of bank names

returned error flag with meaning
    status =  0 - no error
           = -1 - end-of-file
           <  1 - read error
```

By calling

```
status = bosRead (int descriptor, int *jw, void *list)
```

the next BOSIO-record is read and the banks are stored in array "jw". The list "list" in this call may be only of the one-character form. At return the list contains the names of the banks read in.
By calling

[2] Volker Blobel, Pavel Binko, Zbigniew Szkutnik, F - package for input/output, DESY, Germany

```
status = bosWrite (int descriptor, int *jw, void *list)
```

all banks specified in the list "list" are written; since a buffer is used, it is *important* to write finally the last buffer, which is done using an empty list (list = '0'),

```
status = bosWrite (int descriptor, int *jw, '0')
```

5.2 Open, close and rewind

Arguments

```
filename   - file name
flags      - control flags; can be ``r'', ``w'', ``u'' and ``a''
descriptor - file descriptor

returned error flag with meaning
    status =  0 - no error
           <  0 - read error
```

By calling

```
status = bosOpen (void *filename, void *flags, int *descriptor)
```

file with name "filename" and mode "flags" will be opened.

By calling

```
status = bosClose (int *descriptor)
```

file with descriptor "descriptor" will be closed.

By calling

```
status = bosRewind (int *descriptor)
```

file with descriptor "descriptor" will be rewined.

Example: The following code shows the statements for a sequence of operations, in which a record is read, processed and written. After writing all banks of the record are dropped, and after a garbage the process is repeated. The basic array "jw" and the list 'E' is used.

```
   m: status = bosRead(descriptor1,jw,'E');
      if(status == -1) goto exit;            /* end of file */
      ...(processing)
      status = bosWrite(descriptor2,jw,'E');
      if(status < 0) goto ...                 /* error processing */
      bosNdrop(jw,'E');
      bosNgarbage(jw);
      goto m;
exit: status = bosWrite(descriptor2,jw,'0');
      if(status < 0) goto ...                 /* error processing */
      ...
```

6 Changing the length of a bank

Named and work banks are contiguous (without gaps) in storage, and therefore the change of the number of data words of a bank is a distorting operation. However for certain applications this is certainly necessary and should be done in an efficient way, as explained below.

6.1 Work banks

In work banks the word w[id-2] contains NT, the total number of words of the bank. If a work bank is created with ND data words using bosWcreate(), then NT=ND+NHW (NHW - the number of words in the bank header). The value of NT (and not ND) is used in system operations (shift of banks) as the length of the bank. If by a subsequent call of bosWcreate() with the same index ID the number ND of data words is increased, the system has to shift the work bank and also work banks created after the initial creation of the work bank under consideration. The length NT is updated. If by a subsequent call of bosWcreate() the number ND of data words is reduced, only the value of w[id] $\equiv$ ND is changed, while NT is left unchanged. This is a simple and fast operation (which could also be done by the user directly without a subprogram call). If afterwards the number ND of data words is increased again, this is possible without a shift of banks as long as $\mathrm{ND} \leq \mathrm{NT} - \mathrm{NHW}$. If the number ND of data words of a work bank is frequently changed (and the bank is never dropped), the total length will approach the maximum length and there will be only few time-consuming shift operations.

6.2 Named banks

If the number ND of data words of an existing bank is increased by bosNcreate(), the bank will be copied to the gap (free space) and increased in length, the bank at the previous position is dropped. If the bank under consideration is the last named bank (just before the gap), no copy is necessary. In the general case however the copy operation is necessary, because a shift (like in the case of work banks) is not possible for named banks (indices of other banks would change). If the number ND of data words of a named bank is reduced by a call of bosNcreate() by more than NHW words, this operation is possible without a copy; the systems marks the gap between named banks as if it is dropped bank (when a named bank is dropped, the content of w[ind] $\equiv$ ND is changed to $-\mathrm{ND}-\mathrm{NHW}$). This is not possible for a decrease by less than NHW words and the system will copy the bank to the gap, as in the case of an increase of ND.
The conclusion from this discussion is the following: the frequent change of the number of data words ND of named banks should be avoided. This can be avoided by using a work bank, which allows frequent changes in an efficient way.

7 Tables

7.1 BOSIO formats

Data banks in the BOSIO system can lives in 3 different formats: basic format, DAQ format and FPACK format. BOSIO has procedures for conversion between this formats which will be described later. In present section all three formats are described.

Basic format

The structure of BOSIO arrays in Basic format is shown in table 1. Each array has system words in the first 19 words, explained in table 2, followed by a region of fixed length for the indices of named banks, and followed by the named banks. Separated by the unused space (gap), the work banks extends to the end of the array. The table 2 explains the systems data in the first 19 words of each array. Words with default value specified can be changed by the user after the initialization.

Named banks are stored in the low index part of an array. The index "ind" points to the bank, more precisely: the value of "ind" is the index of the word within the array, which contains the number of data words. The structure of a named bank is shown in table 3.

May be in future will be more convenient to use another internal representation for named banks. This representation more closely to DAQ and FPACK formats and can save time for header reordering. The alternative structure of a named bank is shown in table 4.

Work banks are stored in the high index part of the array. The individual identifier "id" is used as an index and has to be a static, to avoid potential problems with certain optimizing compilers. The structure of a work bank is given in table 5.

DAQ format

This format lives in Readout Controllers. Any application program loaded in ROC will handling data using this format (see tables 6 and 7).

index	content
1	array length (4-byte words)
2 ... 19	BOSIO system area
20 ...	named bank
	named bank
	...
	named bank
...	free space (gap)
...	work bank
	work bank
	...
	work bank

Table 1: Structure of an BOSIO array in Basic format

word	content
1	array length
2	check word
3	INDEX OF WORK BANK WITH FORMAT INDICES
4	INDEX OF WORK BANK WITH POINTERS
5	INDEX OF WORK BANK WITH NAMES
6	FIRST INDEX OF NAMED BANK AREA
7	FIRST INDEX OF GAP
8	FIRST INDEX OF WORK BANK AREA
9	NUMBER OF DROPPED WORDS OF NAMED BANKS
10	NUMBER OF DROPPED WORDS OF WORK BANKS
11	INDEX OF LOWEST DELETED NAMED BANK
12	INDEX OF HIGHEST DELETED WORK BANK
13	INDEX OF LINK BANK
14	INDEX OF WORK BANK FOR LIST 'C'
15	INDEX OF WORK BANK FOR LIST 'E'
16	INDEX OF WORK BANK FOR LIST 'R'
17	INDEX OF WORK BANK FOR LIST 'S'
18	INDEX OF WORK BANK FOR LIST 'T'
19	pointer on system structure

Table 2: Systems data

index	content
$ind - ICOL$	the number of columns NCOL
$ind - IROW$	the number of rows NROW
$ind - INAM$	name NAME
$ind - INR$	number NR
$ind - INXT$	index of next bank NXT
ind	the number of data words ND
$ind + 1$	data word 1
$ind + 2$	data word 2
...	...
$ind + ND$	data word ND

Table 3: Structure of named bank

index	content
	empty
ind − INAM	name NAME
	4 spaces
ind − INR	number NR
ind − ICOL	the number of columns NCOL
ind − IROW	the number of rows NROW
	empty
	empty
ind − INXT	index of next bank NXT
ind	the number of data words ND
ind + 1	data word 1
ind + 2	data word 2
. . .	. . .
ind + ND	data word ND

Table 4: Alternative structure of named bank (just for discussion !)

index	content
id − 5	the number of columns NCOL
id − 4	the number of rows NROW
id − 3	name (default is 'WORK')
id − 2	total nr of words in the bank NT
id − 1	pointer to index ID
id	number of data words ND
id + 1	data word 1
id + 2	data word 2
. . .	. . .
id + ND	data word ND

Table 5: Structure of work bank

index	content
1	array length (4-byte words)
2	roc number , counter etc DAQ stuff
3 . . . 10	bank header
11 . . . N1	data
N1+1	empty
N1+2	empty
N1+3 . . . N1+10	bank header
N1+11 . . . N2	data
. . .	. . .
NN+1	empty
NN+2	empty
NN+3 . . . NN+10	bank header
NN+11 . . . NM	data

Table 6: Structure of a BOSIO array in DAQ format

FPACK format

BOSIO uses this format to input/output set of banks from/to FPACK files. This is internal BOSIO format, user never dealing with this format directly. FPACK format is shown in tables 8 and 9.

7.2 List of BOSIO routines

The list of the BOSIO subroutines is shown in table 10.

7.3 Arguments etc

In tables 11, 12 the arguments of BOSIO system programs and extended arguments are given.

7.4 Printout

Bank printout and certain messages (for example a nonzero return code) are printed on standard output device using printf() function.

7.5 Error conditions

Every function returns error condition as function value. User should check this value for each call.

index	content
ind − INAM	name NAME
ind − INR	number NR
ind − ICOL	the number of columns NCOL
ind − IROW	the number of rows NROW
	empty
	format
ind − INXT	index of next bank NXT
ind	number of data words ND
ind + 1	data word 1
ind + 2	data word 2
...	...
ind + ND	data word ND

Table 7: Structure of named bank in DAQ format

index	content
1	array length (4-byte words)
2	accual array length
3	check word
4	format flag
5	record number
6	name1
7	name2
8	run number
9	event number
10	phys. rec. number
11	dataword
12	segment code (0-3)
13	nwords in record
14 ... 23	bank header
24 ... N1	data
...	...
NN ... NN+10	bank header
NN+11 ... NM	data

Table 8: Structure of a BOSIO array in FPACK format

index	content
1	header length
2	name NAME
3	4 spaces
4	number NR
5	the number of columns NCOL
6	the number of rows NROW
7	segment code(0-3)
8	NW (word number in previous bank)
9	ND (word number in this bank)
10	bank format
...	[bank format]
1	data word 1
2	data word 2
...	...
ND	data word ND

Table 9: Structure of named bank in FPACK format

```
int bosInit (int *jw, int nwords) - initialization of the BOS structure
int bosRemove (int *jw)
int bosCtl (char *request, void *argp)
int bosWcreate (int *jw, int *id, int ncol, int nrow) - create work bank
int bosWdrop (int *jw, int *id) - drop work bank
int bosWprint (int *jw, int id) - print work bank
int bosWgarbage(int *jw) - garbage collection for work banks
int bosNcreate (int *jw, char *name, int nr, int ncol, int nrow) - create
int bosNformat (int *jw, char *name, char *fmt) - define bank format
int bosNlink (int *jw, char *name, int nr) - find index of named bank
int bosNdrop (int *jw, char *name, int nr) - drop named bank
int bosNprint (int *jw, char *name, int nr) - print named bank
int bosNgarbage(int *jw) - garbage collection for named banks
int bosLctl (int *jw, char *opt, char *list) - operation with lists
int bosLname (int *jw, int n, char *list) - return n-th name from list
int bosLdrop (int *jw, char *list) - drop banks, specified by list
int bosLprint (int *jw, char *list) - print banks, specified by list

int bosOpen (void *filename, void *flags, int *descriptor)
int bosRead (int descriptor, int *jw, void *list)
int bosWrite (int descriptor, int *jw, void *list)
int bosClose (int descriptor)
int bosRewind (int descriptor)
int bosIoctl (int descriptor, char *request, void *argp)

int mamind (int *jw, int nama)
int listn1(int *jw, char *list, int *jls, int *nls)
int listn2(int *jw, char *list, int *ils, int *jls, int *nls)
int nindex(int *jw, int nr, int nami, int *indi, int *indj)
int namen(int *jw, int nama)
int namez(int *jw, int nama)
```

Table 10: List of BOSIO routines

A Program organisation using BOSIO

A.1 Modular program structure

The BOSIO system is primarily designed for the use in the reconstruction and analysis programs of high energy physics experiments. In these programs usually records or events are processed sequentially and a MAIN program should have a modular structure as shown in the figure below.

```
               Initialization
                     V
+------->--------+
|                V
|       +----------------+
|       |                |
|       |  IO - MODULE   |
|       |                |
|       +----------------+
|                V
|                +--------->----------+
|                |                    |
|       +----------------+            V
|       |                |          STOP
|       |    MODULE  1   |
|       |                |
|       +----------------+
|                V
|                .
|                .
|                |
|       +----------------+
|       |                |
|       |    MODULE  n   |
|       |                |
|       +----------------+
|                V
+-------<--------+
```

The processing loop starts with a call to an IO-module, which performs the read and write operations. Then the different processing modules are called.

A module for data processing can be steered completely by data in banks. A complete program for the sequential processing of data records may consist out of a MAIN program, which calls an input/output module (for example FSEQR) and all the processing modules. A new processing module can be added by adding one line in the MAIN program.

A single processing program module can be defined as a set of subprograms with a steering subroutine, which performs a certain task of data reduction (event reconstruction, event analysis). The concept of a program module requires to have a minimum dependence on other programs, not belonging to the module. It is recommended to use the following general conventions:

- all banks are stored in the basic array IW/RW
- all bank names of the current data (event data) are stored in the list 'E'

It is further recommended to observe the following rule in the design of data banks and processing modules: *"A bank with a given name is created and completely filled in one module only, and no changes to the data are made in modules applied later."*

A module requires input data, preferentially banks with a single name, and will produce output data (results), which preferentially are stored in output banks with a single name. The data flow within a module is shown in the figure below.

```
                       +-----+     +-----+     +-----+
    input banks        |     |     |     |     |     |
                       +-----+     +-----+     +-----+
                           \          |          /
                             \        |        /
                               \      |      /
                                 V    V    V
                             +-----------------+
                             |                 |
                             |     MODULE      |
                             |                 |
                             +-----------------+
                                 V    V    V
                               /      |      \
                             /        |        \
                           /          |          \
                       +-----+     +-----+     +-----+
    output banks       |     |     |     |     |     |
                       +-----+     +-----+     +-----+
```

In the example below it is assumed, that the module requires input data in banks with the name BKIN, and produces new data, which are stored in the bank BKUT. A data reduction module could have a structure as shown in the listing below.

```
      SUBROUTINE MODUL
      COMMON/BCS/ISPACE(5),IW(1000)
      REAL RW(1000)
      EQUIVALENCE (IW(1),RW(1))
      INTEGER NAMI/0/
*     at first entry determine name-indices and define format
*     for output bank
      IF(NAMI.EQ.0) THEN
         NAMI=NAMIND('BKIN')
         NAMU=NAMIND('BKUT')
         CALL BKFMT('BKUT','3I,2F')
      END IF
*     immediate return, if either output bank already exists,
*     or input bank does not exist
      IF(IW(NAMU).NE.0.OR.IW(NAMI).EQ.0) GOTO 100
*     add name of output bank to list 'E'
      CALL BLIST(IW,'E+','BKUT')
*     prepare loop on all input banks
      INDI=NAMI+1
   10 INDI=IW(INDI-1)
      IF(INDI.EQ.0)  GOTO 100
      NR=IW(INDI-2)
*     create output bank with same number as input bank
      INDU=NBANK('BKUT',NR,10)
      IF(INDU.EQ.0) GOTO 100
*     data reduction for data in bank (BKIN,NR), index INDI
*     result will be stored in output bank (BKUT,NR), index INDU
*     ...
      GOTO 10
  100 RETURN
      END
```

In the above program the name-indices for input and output banks are determined at the first entry to allow fast access to the banks. In addition the format of the output bank is defined. Then it is checked, wether computation is necessary. If either the output banks already exist, or the input banks do not exist, computation is not necessary or not possible and the module returns immediately to the calling program. If the user wants to repeat the computation (for example, if the module has been improved), he can drop the output banks before the call of the module by the statement

```
CALL BDROP('IW','BKUT')
```

and then the module will create new banks. If computation has to be done in the module, the name of the output banks will be added to the list 'E', in order to get the banks written together with the other data banks. Then a loop is executed over all input banks. Note that the banks numbers do not have to be consecutive.

Options in FSEQR

Subprogram FSEQR may be used directly as the IO-module or may serve as a model for a more specialized version. In high energy physics experiments one record usually contains the data of one event (interaction of elementary particles), which is characterised by a run and an event number. Subroutine FSEQR contains options to select certain events or runs or to ignore certain runs. A fast record skipping method is used. A necessary condition for the use of the option is the storage of the run and event number in the first (header) bank, with a fixed name and number, of all records (note that in writing records the order of banks in a record is defined by the list of names, not by the order of bank creation). If a records does not contain the specified bank as first bank, the record will be accepted.
The user can define the relevant parameters for the bank containing run and event number by a call of FSEQH.

```
CALL FSEQH(NAME,NR,IRUN,IEVT)..........define header bank
```

The meaning of the arguments is: the bank (NAME,NR) will contain the run number at word IRUN and the event number at word IEVT. The default assumption is equivalent to the call

```
CALL FSEQH('HEAD',0,2,3)
```

Possible options are

- select certain event numbers (SEVT)
- select certain run numbers (SRUN)
- ignore certain run numbers (IRUN).

The priority of the options is in the order given; if for example the option SEVT is used, the options SRUN and IRUN cannot be used. The use of the options require banks with names SEVT, SRUN and IRUN, which most easily are defined on card image records and read with the subroutine BREADC (see next chapter). The content of the card image records is as follows:

```
SEVT  nrun   nevt   nevt   nevt ...          ! selected events
SRUN  nrun   nrun   nrun1 -nrun2 ...         ! selected runs
IRUN  nrun   nrun   nrun   nrun1 -nrun2 ... ! runs to be ignored
```

Any number of these card image records is allowed. The SEVT card image record has to contain the run number as first integer, followed by the event numbers. The SRUN and IRUN card image records contain the run numbers. If a whole range NRUN1...NRUN2 of run numbers is required, this can be specified in the form NRUN1 -NRUN2 (upper limit of range given as a negative number).
If the user wants to skip the run of the current event he may call BSEQS.

```
CALL FSEQS............................skip current run
```

An example of a MAIN program, where all options are used, is given below.

```
      COMMON/BCS/ISPACE(5),IW(5000)
      CALL BOS(IW,5000)
      CALL BREADC                       read data cards
      CALL FSEQH('HEAD',0,1,2)          define header bank
      CALL FSEQP(5,1000)                define limit parameters

      LUNR = 1
   10 CALL FSEQR(LUNR,IRET)
      IF(IRET.LT.0) GOTO 100
*     ...
      CALL MODULE
      CALL FSEQW(3)
      IF(...) CALL FSEQW(4)
      IF(...) CALL FSEQW(-3)
*
      IF(...) CALL FSEQS                skip run
      IF(...) CALL FSEQE                force end-condition
*     ...
      GOTO 10
  100 STOP
      END

                                        card image records
SRUN  22345   21234  -21260   21261           ! single beam runs
SRUN  30100   30200   30300  -30399           ! cosmic runs
```

A.2 Module steering(H1 stuff - temporary here)

A module should contain the logic for the initialization of the module and other special situations (run start or run end for example). The situation can be determined from the data in banks and is supported by subprogram FSEQR, if the header bank containing run and event number is defined. FSEQR provides some data for the steering of a module in a common /BOSMDL/, explained at the end of this section. Further steering information is provided by the use of the subprogram SMODUL.

SMODUL has to be called as the first executable statement in each module with the module name (called 'SUBNAM' here) as argument:

```
      SUBPROGAM SUBNAM
      COMMON/BCS/  ...
      CALL SMODUL('SUBNAM')
      ...
  100 RETURN
      END
```

The subprogram SMODUL has three functions: book keeping on modules, steering of histogram calls and steering of debug printout.

Book keeping. At a call of SMODUL a bank (BOOK,0) is either created or extended; the name of the module is recorded in two consecutive words (A format) and the date is recorded in a third word (integer). If the name of the module is already contained in the bank BOOK, the date of the previous application of the module is returned (IDATEL) in the steering common and the date within the bank BOOK is overwritten by the actual date.

Steering of debug printout. Usually a module contains some debug printout statements. This debug printout should be controlled by the flag IDB in the steering common, with the following meaning:

value	meaning
IDB = 0	no printout
IDB = 1	minimum printout
IDB = 2	more printout
IDB = 3	even more printout

If a user of a module wants debug printout for the following ICOUNT calls of the module with a value IDEB, he has to call

```
      CALL SETDB('SUBNAM',IDEB,ICOUNT)
```

Then in the following ICOUNT calls of the module SUBNAM the debug flag IDB is set to the argument value IDEB, and to zero for the following calls.

The content of the steering common/BOSMDL/ is as follows:

```
*     STEERING COMMON
      LOGICAL       BEGJOB,ENDRUN,BEGRUN,REVENT,ENDJOB,OTHDAT
      COMMON/BOSMDL/BEGJOB,ENDRUN,BEGRUN,REVENT,ENDJOB,OTHDAT,
     +              LCCRUN,NCCRUN,NEVENT,
     +              IHA,IBS,IDB,IDATEL

Logical flags, set at the return from FSEQR (require FSEQH):
         true ...          (false otherwise)
BEGJOB        at first record only
ENDRUN        at the first event after a run (last run is = LCCRUN)
BEGRUN        at first event of a run (run is NCCRUN)
REVENT        if header bank found (event read in)
ENDJOB        after end-of-data (no record read in)
OTHDAT        if no header bank found (no event)

Numerical data, set at the return from FSEQR:

LCCRUN        run number of previous event
NCCRUN        current run number
NEVENT        current event number

Numerical data, set at the return from SMODUL:

IHA           offset for histograms (multiple of 100)
IBS           offset for BOSIO-histograms
IDB           debug flag (0 means no debug printout)
IDATEL        date, if module already applied to event (else 0)
```

B Using BOSIO from FORTRAN

Arguments:

```
DSN ..................... * ........data set name
FMT ..................... * ........format of named bank
ID ................................index of a work bank (in a common)
IND ...............................index of a named bank
IW/JW .............................BOSIO-array
LEG ..................... *2 .......'LE' or 'EQ' or 'GE'
LIST .................... * ........list, defining set of banks
LUN ...............................logical I/O-unit number
IOMODE.................... * ........I/O mode, 'FORT' or 'EPIO'
IRECL .............................nr of words in a da record
N .................................sequ. nr of a name in a list
NAME .................... *4 .......name of a bank
NBLK ..............................buffer size for IO
NCOL ..............................number of columns in a bank
NROW ..............................number of rows in a bank
NDIM ..............................dimension of BOSIO-array
NR ................................number of named bank
NREC ..............................number of records for da-ds
NRMAX .............................largest nr of a named bk (BWIND)
OPT ..................... *2 .......option selected in BLIST
PASSWD .................. * ........password for da ds
*ret...............................alternate return
                           * means character string

----------------------------------------------------------------------


General IO program and module control program:

CALL FSEQP(NSEC,NLIM)............parameter for general IO program
CALL FSEQH(NAME,NR,IRUN,IEVT).....specify header bank
CALL FSEQR(LUNR,IRET)............general IO program/input of rec
CALL FSEQW(LUNW).................set flag for output
CALL FSEQS.......................skip current run
CALL FSEQE.......................end condition

CALL SETDB('subnam',IDEB,ICOUNT)..set debug flag
CALL SMODUL('subnam')............start module


*     ***** Model  M A I N   program *******

*     declare BOSIO commons and initialize BOSIO
*     BASIC BOSIO common for data processing
      PARAMETER (LENBCS=50000,LENHCS=10000)
      COMMON/BCS/ISPACE(5),IW(LENBCS)
      REAL       RW(LENBCS)
      EQUIVALENCE(RW(1),IW(1))
*     Common /HCS/ for BOS histograms
      COMMON/HCS/KSPACE(5),KW(LENHCS)
      CALL   BOS(IW,LENBCS)
      CALL   BOS(KW,LENHCS)
*     read data cards
      CALL BREADC
*     define parameter: CALL FSEQP(NSEC,NLIM)
*     NSEC = nr of seconds> stop NSEC seconds before reaching timelimit
```

```
*     NLIM = nr of records> stop after NLIM records. 0 = infinity
      CALL FSEQP(2,0)
*     run and event number in 'HEAD',0 at words 2 and 3
      CALL FSEQH('HEAD',0,2,3)
*     select IO mode, if necessary
C     CALL BUNIT(UNIT,IOMODE,NBLK)
      LUNR=1
*     event processing loop
   10 CALL FSEQR(LUNR,IRET)
      IF(IRET.LT.0) GOTO 100
*     call modules here
      CALL SUBNAM
      GOTO 10
  100 STOP
      END
      SUBROUTINE SUBNAM
*     Standard BOS common
      COMMON  /BCS/ISPACE(5),IW(1000)
      REAL         RW(1000)
      EQUIVALENCE (RW(1),IW(1))
*     Common with steering data:
      LOGICAL       BEGJOB,ENDRUN,BEGRUN,REVENT,ENDJOB,OTHDAT
      COMMON/BOSMDL/BEGJOB,ENDRUN,BEGRUN,REVENT,ENDJOB,OTHDAT,
     +              LCCRUN,NCCRUN,NEVENT,
     +              IHA,IBS,IDB,IDATEL
*     comment to common above
*     Logical flags, set at the return from FSEQR
*             true ...          (false otherwise)
*     BEGJOB        at first record only
*     ENDRUN        at the first event after a run (last run is= LCCRUN)
*     BEGRUN        at first event of a run (run is NCCRUN)
*     REVENT        if header bank found (event read in)
*     ENDJOB        after end-of-data (no record read in)
*     OTHDAT        if no header bank found (no event)
*
*     Numerical data, set at the return from FSEQR:
*
*     LCCRUN        run number of previous event
*     NCCRUN        current run number
*     NEVENT        current event number
*
*     Numerical data, set at the return from SMODUL:
*
*     IHA           offset for histograms (multiple of 100)
*     IBS           offset for BOS-histograms
*     IDB           debug flag (0 means no debug printout)
*     IDATEL        date, if module already applied to event (else 0)
*     ...
*     module steering by SMODUL
      CALL SMODUL('SUBNAM')
*
      IF(BEGJOB) THEN
*        at first record only
      END IF
      IF(ENDRUN) THEN
*        at the first event after a run (last run is= LCCRUN)
      END IF
```

```
      IF(BEGRUN) THEN
*        at first event of a run (run is NCCRUN)
      END IF
      IF(REVENT) THEN
*        header bank found (event read in) - process data
      END IF
      IF(ENDJOB) THEN
*        after end-of-data (no record read in)
      END IF
  100 RETURN
      END

Data cards for run and event selection (requires FSEQH):

SEVT  nrun   nevt   nevt   nevt  ...   ! selected events
SRUN  nrun   nrun   nrun1 -nrun2 ...   ! selected runs
IRUN  nrun   nrun1 -nrun2  nrun  ...   ! runs to be ignored

----------------------------------------------------------------------
```

B.1 General input/output subprogram

The general subprogram for sequential input/output FSEQR contains all calls to read a data set sequentially, to write the records on a sequential data set (optionally), and also to drop the banks of the previous record and to make a garbage collection before a new record is read. It may either be used directly or serve as a model for a similar, but specialized program for a certain application. The following conventions apply:

- The basic array IW/RW in common /BCS/ is used to store the banks of the read records.
- The list 'E' is used to keep the names of the banks.
- Records with read errors are skipped.

FSEQR allows to use one input unit and up to four output units. The mode for each of these units has to be selected by the user *before* the first call of FSEQR.
The subprogram FSEQR has several options, described below.

Input

```
      CALL FSEQR('BOSINPUT',IRET)..........................read from BOSINPUT
```

The return argument IRET has the value 1, if a record has been read. If the end-condition is reached in FSEQR (for example by an end-of-file on the input unit), the argument IRET is 0 at return and will be -1, if FSEQR is called once more. The IF-statement in the MAIN program above assumes one pass through the processing module after the end-condition. This will not be done if the test is for IRET.LE.0.

Parameter

By default FSEQR will stop reading 2 seconds before the timelimit or at an end-of-file on the input data set. These limits are defined by parameters, which can be changed by a call of FSEQP before the first call of FSEQR.

```
      CALL FSEQP(NSEC,NLIM)................................define parameter
```

Reading will stop NSEC seconds before the time limit or after NLIM records, where NLIM = 0 means no limit. If the user wants to stop reading at a certain condition, he may call FSEQE at any time.

```
      CALL FSEQE.....................................require end condition
```

At the next call of FSEQR the end-condition is assumed.
User can skip a run using FSEQS subroutine.

```
      CALL FSEQS............................................skip present run
```

Output

Default is no output. Output can be done to up to four different units. A call to FSEQW sets an output flag; actual output is done at the next call of FSEQR. If the end-condition is reached in FSEQR, the buffers for used output units will be written.

```
CALL FSEQW('BOSOUTPUT').................................set output flag
```

Example: An example for the minimal version of a MAIN program for reading a data set and processing (assumed to be done in subprogram MODULE) is given below.

```
      COMMON/BCS/ISPACE(5),IW(5000)
      CALL   BOS(IW,5000)
      ...
   10 CALL FSEQR('BOSINPUT',IRET)
      IF(IRET.LT.0) GOTO 100        ! end of data
      CALL MODULE                   ! processing
C      CALL FSEQW('BOSOUTPUT')       ! if output is needed
      GOTO 10
  100 STOP
      ...
      END
```

In this MAIN program all records from BOSINPUT are read, after each input of a record the subprogram MODULE is called for processing.

type	argument	explanation

Table 11: Arguments of BOSIO subprograms

content of jw1	explanation	subprogram
=1	additional words not set to zero	bosNcreate(), bosWcreate(),...
=2	move bank to gap of free space	bosNcreate(), bosWcreate()
=3	create bank and drop all other banks of the same name	bosNcreate(), bosWcreate()
=1	drop banks after write	

Table 12: Extended arguments in jw1 in SYS structure