E10-2001-199

A.Yu.Isupov

CONFIGURABLE DATA AND **CAMAC** HARDWARE REPRESENTATIONS FOR IMPLEMENTATION OF THE **SPHERE DAQ** AND OFFLINE SYSTEMS

1 Introduction

Data acquisition system's implementation for the SPHERE experiment [1] based on the distributed portable data acquisition and processing system *qdpb* [2] now. This paper describes code, added to the *qdpb*'s framework to support an experimental setup description: experimental data and CAMAC hardware representation schemes, both configurable for RUN[1] independence purposes.

Through the following text the file and software package names are highlighted as *italic text*, C language constructions and reproduced "as is" literals – as `typewriter text`. Reference to manual page named "qwerty" in the 9th section printed as ***qwerty(9)***, reference to section in this report – as 2.2.1. Subjects of substitution by actual values are enclosed in the angle brackets: `<run_name>`. All mentioned trademarks are properties of its respective owners.

2 Experimental data configuration scheme

An experimental data representation scheme is designed to provide runtime configurable description of such data, so allows to implement an SPHERE DAQ and offline utilities (see, f.e., 2.4) in a RUN independent manner. It implemented on base of the following program objects:

- universal data containers (so called cells) implementation (see 2.1),
- universal variables representation (so called knvar) (see 2.2.1).
- universal functions representation (so called knfun) (see 2.2.2).
- universal presenter objects (so called knobjs) implementation (see 2.2.3, 2.2.4).

Also provided interface for reading of so called RUN configuration file in the ***RUN.conf(5)*** format (see 2.3).

2.1 Universal data containers – cells

The *cell.conf* is a configuration file in so called ***cell.conf(5)*** format for array of so called cells, universal data containers, used by some utilities of the SPHERE DAQ and *offline* systems. This file consists of zero or more lines, delimited by newline symbols. Lines may be:

- comment lines,
- empty lines, and
- data lines,

where newline may be escaped by backslash for line continuation. Comment lines (lines with "#" in the first position) and empty lines are ignored.

Other lines must be data lines. Data lines, concatenated with all its continuations, contains four fields, separated by space(s) or tab(s). First three fields can't contains space(s) or tab(s), fourth can.

[1](in the wide meaning) – accelerator run.

First field (`name`) of the data line is a string. It represents name, under which respective cell will be known in DAQ or offline utility.

Second field (`type`) is a string (f.e. "`type_UChar`"), represents required type of cell result. Types are `#define`'d in the *pdata.h* header file as follows:

```
#define type_UChar      1
#define type_UShort     2
#define type_ULong      4
#define type_Double     5
#define type_Float      6
#define type_Char       11
#define type_Short      12
#define type_Int        13
#define type_Long       14
#define type_String     20
```

Third field (fill dependence) is a string, represents event kind in form `DATA_<EVENT_KIND>`[2], for which cell calculations must be performed. Possible event kinds are `#define`'d in the *pack_types.h* header file. Also permitted two special event kinds: `PROG_BEG` and `PROG_END`.

Fourth field (program) is a string, beginning after third field and ending at end of data line, represents "program" for cell result calculations. It is a right part of C expression (without assignment), where operands may be a:

- constant value,
- already initialized cell (including current cell),
- known variable (see 2.2.1), or
- other valid expression.

Following operations are implemented:

unary: "+", "-", "~", "!"

binary: "+", "-", "*", "/", "==", "!=", "<", ">", "<=", ">=", "&&", "||", "<<", ">>", "&", "|", "^"

ternary: "? :"

Calculations performed in order, defined by C operation priorities, and may be changed by operands grouping in parentheses, "`(`" and "`)`". Following functions are implemented (with one argument, if not noted otherwise): "`sin`", "`cos`", "`tan`", "`asin`", "`acos`", "`atan`", "`sinh`", "`cosh`", "`tanh`", "`asinh`", "`acosh`", "`atanh`", "`exp`", "`expm1`", "`log`", "`log10`", "`log1p`", "`pow`" (with two arguments), "`sqrt`", "`cbrt`", "`fabs`", "`erf`", "`erfc`", "`j0`", "`j1`", "`y0`", "`y1`".

So ***cell.conf(5)*** describes for each defined calculation cell a structure `cell` with:

- calculation cell name;
- program of result calculation;
- required type of result;

[2]`<EVENT_KIND>` need to be substituted by uppercased name of event kind (for example, "`CYC_BEG`", "`DAT_0`", "`CYC_END`", etc.)

• event's (in particular – packet's) kind, which causes result calculation.

Result will be calculated, if current value of the global counter, corresponding to cell's event kind, is greater than saved value. Global counters, corresponds to event kinds PROG_BEG and PROG_END, need to be incremented near begin and end of utility's main() function.

```
#include <cell.h>
int cell_cfg(char *cellfile, cell *cells, u_long *counters,
        size_t cnt_len)
void *get_cell(cell *cells, char *str, int len, u_char *type)
int cell_calc(cell *gCell)
inline u_char str2vartypes(char *tok, union pres_arg *res,
        union pres_arg *ptr)
void CELL_CALC_LOOP(void)
void CELL_DEBUG_LOOP(void)
#include <parser.h>
int scanner(char *str, struct token *tokens, unsigned *tlen)
int exp_parser(struct token *tokens, unsigned tlen,
        struct action *actions, unsigned *alen)
#include <actions.h>
double execute(struct action *actions, unsigned size)
```

cell_cfg, get_cell, cell_calc, str2vartypes, scanner, exp_parser, execute, CELL_CALC_LOOP, CELL_DEBUG_LOOP, – routines intended for cell manipulations.

cell_cfg() reads file *cellfile in the ***cell.conf(5)*** format (see above) and fills already allocated array of cell structures, defined in the *cell.h* header file, pointed by *cells, and global counters array with size cnt_len, pointed by *counters.

get_cell() searches for cell with name *str of length len in array of cell structures, pointed by *cells, fills *type by correct type of cell result, and returns correct pointer to cell result. Possible result types are #define'd in the *pdata.h* header file.

cell_calc() calculates result of proper type for cell *gCell and stores it in gCell->res.

str2vartypes() returns type number (#define'd in the *pdata.h* header file) for string *tok, and if both pointers *res and *ptr are non equal to NULL, directs *ptr of correct type into *res.

Macro CELL_CALC_LOOP() performs result calculation, if fill dependence is expired, for all initialized (by cell_cfg()) cells in the global array gCell of its.

Macro CELL_DEBUG_LOOP() prints cell member's values for all initialized (by cell_cfg()) cells in the global array gCell of its.

scanner() breaks string *str, represents C expression, stores resulted tokens in array of token structures, defined in the *parser.h* header filepointed by *tokens, and fills *tlen by obtained tokens number.

exp_parser() reads tlen tokens from array of token structures, pointed by *tokens, fills array of action structures, defined in the *actions.h* header file– "program", pointed by `*actions` accordingly, and fills `*alen` by total actions number ("program" length).

`execute()` returns result of calculations by "program" of length `size`, represented by array of `action` structures, pointed by `*actions`.

`cell_cfg()` returns 0 on success and > 0 if error occurred.

`cell_calc()` returns 0 on success and 1 if unknown type of result required.

`str2vartypes()` returns `(u_char)(-1)` if string does not correspond to any known type.

2.2 Universal presenter object – knobj

2.2.1 Universal variables representation – knvar

```
#include <knvar.h>
void *get_variable(knvar *var, char *str, int len, u_char *type)
void add_variable(knvar *var, char *str, u_char type, void *ptr)
void KNVAR_INIT(knvar *var)
```

`get_variable`, `add_variable`, `KNVAR_INIT` – routines intended for handling with so called known variables.

`get_variable()` searches for known variable by its name `*str` of length `len` in the `*var` array of knvar structures, defined in the *knvar.h* header file as follows:

```
typedef struct {
        char *tvar;      /* pointer to name for exp. data variable */
        u_char type;     /* value type */
        void *ptr;       /* pointer to value itself */
} knvar;
```

fills `*type` by known variable type (`#define`'d in the *pdata.h* header file), and returns pointer to corresponding `knvar` value. `*var` must be terminated properly (i.e. `tvar` in the first unused `knvar` structure must be equals to `NULL`).

`add_variable()` adds variable of type `type` with name `*str` and value, pointed by `*ptr`, into tail of array of `knvar` structures, pointed by `*var`, and `NULL` terminates `*var` properly.

Macro `KNVAR_INIT()` terminates `var` properly (see above).

`get_variable()` returns `NULL` if variable not found.

So, provided tools for: any variable in DAQ or offline utility be registered in the `knvar` structure (as so called "known variable", see above).

2.2.2 Universal functions representation – knfun

```
#include <knfun.h>
knfun *get_func(knfun *funcs, char *str, int len)
void add_func(knfun *funcs, char *str, func_t create, func_t fill,
        func_t clean, func_t remove)
void KNFUN_INIT(knfun *funcs)
```

get_func, add_func, KNFUN_INIT – routines intended for handling with so called known functions.

get_func() searches for known function member by its name *str of length len in the *funcs array of knfun structures, defined in the *knfun.h* header file as follows:

```
typedef struct {
#define FUNNAME_LEN     32
        char    name[FUNNAME_LEN];
        func_t  create;
        func_t  fill;
        func_t  clean;
        func_t  remove;
} knfun;
```

(for func_t definition see 2.2.4), and returns pointer to corresponding knfun structure. *funcs must be terminated properly (i.e. name in the first unused knfun structure must be an empty string, "").

add_func() adds known function with name *str into tail of array of knfun structures, pointed by *funcs, fills this member by set of functions create, fill, clean, remove, and NULL terminates *funcs properly.

Macro KNFUN_INIT() terminates funcs properly (see above).

get_func() returns NULL if function not found.

So, provided tools for: any function in DAQ or offline utility be registered in the knfun structure (as so called "known function", see above).

2.2.3 Knobjs configuration files

The ***knobj.conf(5)*** is a configuration file for array of so called known objects (knobjs), universal presenter objects, used by some utilities of the SPHERE DAQ and *offline* systems. This file consists of zero or more lines, delimited by newline symbols. Lines may be a:

- comment lines,
- empty lines, and
- data lines,

where newline may be escaped by backslash for line continuation. Comment lines (lines with "#" in the first position) and empty lines are ignored.

Other lines must be data lines. Data lines, concatenated with all its continuations, contains nine fields, separated by space(s) or tab(s) and not contains it.

First field (name) of the data line is a string. It represents name, under which respective object will be known in DAQ or offline utility.

Second field (creator parameters) is a string, represents one or more parameters for create function of such known object. Parameters separated by ";". Each parameter is a:

- constant value (in particular C string without surrounding double quotes """ and with mandatory type type_String definition),

- known variable (see below), or
- initialized cell name (see 2.1),

optionally followed by "," and its type in a string representation (possible types the same as for second field of data line of the ***cell.conf(5)***, see 2.1). Here is an example of the second field, contains all mentioned parameter flavours, respectively: "`13;qqq,type_String;aaa,type_UShort;Cell0001`". Field of kind "`-`" or "`-,-`" means parameters not required.

Third field (`type`) is a string, represents type of respective known object. Knobj type dictates the knobj functionality. Names (`knfun[].name`) of creator, filler, cleaner and remover known functions are constructed from knobj type (see below).

Fourth field (filler parameters) is a string, represents one or more parameters for `fill` function of such known object. Syntax the same as for second field, see above.

Fifth field (fill dependence) is a string, represents event kind in form `DATA_<EVENT_KIND>`, for which `fill` function of respective known object must be called. Possible event kinds are `#define`'d in the *pack_types.h* header file.

Sixth field (fill condition) is a string, represents fill condition of such known object. Syntax the same as for second field (see above), but sixth field can contain only one parameter, which can't be constant value, f.e. "`flag,type_UChar`".

Seventh field (clean dependence) is a string, represents event kind in form `DATA_<EVENT_KIND>`, for which clean function of respective known object must be called. Syntax the same as for fifth field (see above). Also permitted one special event kind: `NEVERMORE`, which means clean need not at all.

Eighth field (cleaner parameters) is a string, represents one or more parameters for `clean` function of such known object. Syntax the same as for second field, see above.

Ninth field (remover parameters) is a string, represents one or more parameters for `remove` function of such known object. Syntax the same as for second field, see above.

Second, fourth, sixth, eighth and ninth fields are optional, so may be omitted at the end of data line and replaced by "-" within it.

So ***knobj.conf(5)*** describes for each defined known object a structure `knobj` with:

- known object name;
- known object type;
- parameters for known object's creator, filler, cleaner, and remover;
- event's (in particular – packet's) kinds, which causes known object's fill and clean;
- fill condition.

Creator, filler, cleaner, and remover function names constructed by catenating the known object type and strings "`create`", "`fill`", "`clean`", and "`remove`", respectively, so ones can be founded in array of `knfun` structures.

The ***clean.conf(5)*** is a cleaning list for array of known objects, used by some utilities of the SPHERE DAQ and *offline* systems. This file consists of zero or more lines, delimited by newline symbols. Lines may be a:

- comment lines,

- empty lines, and
- data lines,

where newline may be escaped by backslash for line continuation. Comment lines (lines with "#" in the first position) and empty lines are ignored.

Other lines must be data lines. Data lines contains one field, not contains space(s) or tab(s).

First field (name) of the data line is a string. It represents name of already initialized known object, which need to be cleaned, within DAQ or offline utility.

2.2.4 Knobjs program interface

```
#include <knobj.h>
int cond_check(struct pres_act *cond, u_char type)
int pres_cfg(char *presfile, knobj *knobjs, u_long *counters,
        size_t cnt_len)
int clean_cfg(char *cleanfile, knobj *knobjs)
knobj *get_knobj(knobj *knobjs, char *str, int len)
void KNOBJ_CREATE_LOOP(void)
void KNOBJ_FILL_CLEAN_LOOP(void)
void KNOBJ_CLEAN_LOOP(void)
void KNOBJ_COND_CLEAN_LOOP(void)
void KNOBJ_REMOVE_LOOP(void)
```

cond_check, pres_cfg, clean_cfg, get_knobj, KNOBJ_CREATE_LOOP, KNOBJ_FILL_CLEAN_LOOP, KNOBJ_CLEAN_LOOP, KNOBJ_COND_CLEAN_LOOP, KNOBJ_REMOVE_LOOP – routines intended for handling with so called known objects.

cond_check() returns true (1) if cond->args[0] of type type is true (!0), and false (0) if cond->args[0] of type type is false (0).

pres_cfg() reads file *presfile in the ***knobj.conf(5)*** format (see above) and fills already allocated *knobjs array of knobj structures, defined in the *knobj.h* header file as follows:

```
struct dependence {
        u_long *curcnt; /* pointer to global packet counter */
        u_long oldcnt;  /* its previous saved value */
};
union pres_arg {
        u_char  u_charv;        /* u_char value */
        u_char  *u_charp;       /* u_char ptr */
        u_short u_shortv;       /* u_short value */
        u_short *u_shortp;      /* u_short ptr */
        u_int   u_intv;         /* u_int value */
        u_int   *u_intp;        /* u_int ptr */
        u_long  u_longv;        /* u_long value */
        u_long  *u_longp;       /* u_long ptr */
        double  doublev;        /* double value */
        double  *doublep;       /* double ptr */
        float   floatv;         /* float value */
        float   *floatp;        /* float ptr */
        char    charv;          /* char value */
        char    *charp;         /* char ptr */
        short   shortv;         /* short value */
        short   *shortp;        /* short ptr */
        int     intv;           /* int value */
        int     *intp;          /* int ptr */
        long    longv;          /* long value */
        long    *longp;         /* long ptr */
};
typedef void *  (*func_t)(union pres_arg *, void *);
struct pres_act {
        func_t          fun;
#define PRESARGC_MAX    32      /* ntfunc.c need 24 now... */
        union pres_arg  args[PRESARGC_MAX];
};
typedef struct {
        u_char                  flag;
#define OBJNAME_LEN     32
        char                    name[OBJNAME_LEN];
        struct  pres_act        create;
        void                    *pres;
        struct  pres_act        remove;
        struct  pres_act        fill;
        struct  dependence      fdep;
        struct  pres_act        condfill;
        u_char                  condtype;
        struct  pres_act        clean;
        struct  dependence      rdep;
} knobj;
```

and global counters array with size `cnt_len`, pointed by `*counters`.

`clean_cfg()` reads file `*cleanfile` in the ***clean.conf(5)*** format (see 2.2.3) and sets to `(u_long)(-1)` the `rdep.oldcnt` field of each initialized (by `pres_cfg()`) `knobj` structure from array of it, pointed by `*knobjs`.

`get_knobj()` searches for known object member by its name `*str` of length `len` in the `*knobjs` array of `knobj` structures, and returns pointer to corresponding `knobj` structure. `*knobjs` must be initialized properly (i.e. `flag` in the first unused `knobj` structure must have zero value).

Macro `KNOBJ_CREATE_LOOP()` calls `create.fun` member for all initialized (by `pres_cfg()`) known objects in the global array `knobjs` of its.

Macro KNOBJ_FILL_CLEAN_LOOP() calls clean.fun member, if clean dependence is expired, and fill.fun member, if fill dependence is expired and fill condition check success, for all initialized (by pres_cfg()) known objects in the global array knobjs of its.

Macro KNOBJ_CLEAN_LOOP() unconditionally calls clean.fun member and resynchronizes fill and clean dependencies for all initialized (by pres_cfg()) known objects in the global array knobjs of its.

Macro KNOBJ_COND_CLEAN_LOOP() calls clean.fun member, if clean dependence modified by clean_cfg(), and resynchronizes clean dependence for all initialized (by pres_cfg()) known objects in the global array knobjs of its.

Macro KNOBJ_REMOVE_LOOP() calls remove.fun member for all initialized (by pres_cfg()) known objects in the global array knobjs of its.

cond_check() returns -1 if requested type type is unknown.

pres_cfg() returns 0 on success and > 0 if error occurred.

clean_cfg() returns 0 on success and > 0 if error occurred.

get_knobj() returns NULL if known object not found.

Known function may be a:

- creator,
- filler,
- cleaner, or
- remover

for some known object, represented by knobj structure.

2.3 RUN configuration file

The <run_name>*.conf* is a configuration file for SPHERE DAQ and *offline* systems in so called ***RUN.conf(5)*** format, represents RUN <run_name>. It consists of zero or more lines. Comment lines (lines with "#" in first position) and empty lines are ignored. Other lines may be a:

- chapter header lines,
- chapter's member lines, and
- chapter trailer lines.

All contains one or more fields, separated by space(s) or tab(s).

Chapter header line contains in the first field the event kind in a string representation: "DATA_<EVENT_KIND>", beginning from first position. Next fields (if exists) meaning depends on DAQ or offline utility, which interprets ***RUN.conf(5)*** file.

After chapter line must be present exactly NMEMB_<event_kind>[3] (#define'd in the *e*<run_name>*.h* header file) number of member lines. First field of member line (flag) contains ASCII string, its possible values #define'd in the *pdata.h* header file as follows:

[3]<event_kind> need to be substituted by name of event kind (for example, "cyc_beg", "dat_0", "cyc_end", etc.)

```
#define S_NULL           "0"      /* variable not handled at all */
#define S_HANDLE         "1"      /* variable will be handled */
```

Generally nonzero means, that respective data field need to be processed, and zero – that need not. Second field (tvar) is a string. It represents short name, under which respective data field may be known in DAQ or offline utility (so called "known variable" name, knvar.tvar, see 2.2.1). Next fields (if exists) meaning depends on DAQ or offline utility, which interprets ***RUN.conf(5)*** file.

For each <event_kind> must be only one construction, beginning from chapter header line, contains some member lines, and optionally ending by chapter trailer line. If some kind of event does not described at ***RUN.conf(5)*** file, variables from it will not be processed by DAQ and offline utilities.

So, RUN configuration file in the ***RUN.conf(5)*** format sets flags and names for experimental data fields, so (partially) initializes the pdata (flag and tvar members, see *pdata.h* header file), and knvar (*tvar member, see 2.2.1) structures.

The following program interface to RUN configuration file is provided

```
#define <run_name>
#include <eRUN.h>
#include <knvar.h>
int run_cfg(char *conffile, pdata **pdat, knvar *knvars)
```

run_cfg() reads RUN configuration file *conffile in the ***RUN.conf(5)*** format (see above) and fills already allocated pdat structures (see *pdata.h* header file) and array of knvar structures (see 2.2.1), pointed by *knvars, if it not equals to NULL.

run_cfg() returns 0 on success, and > 0 if error occurred.

The run_cfg() does not set errno status for its internal errors. The run_cfg() may also fail and set errno for any of the errors, specified for the functions ***fopen(3)*** and ***fgets(3)***.

2.4 *offline* utilities

In the current implementation of the SPHERE *offline* system the following utilities are provided:

- the ***ufill(1)*** fills HBOOK/ROOT Ntuples and histograms,
- the ***calibtof(1)*** calibrates SPHERE TOF data,
- the ***ckdata(1)*** checks SPHERE experimental data consistency.

2.4.1 *ufill(1)* utility

ufill(1) utility destined for building HBOOK [3] Ntuples/H[12] histograms or ROOT [5] TNtuples/TH[12] histograms from experimental data, produced by SPHERE DAQ system in ***packet(3)*** format, see [2], or in old (pre–*qdpb*) formats (see also ***get_data(3)***).

```
ufill [-c{-|<runconffile>}] [-s{-|<cellconffile>}]
        [-k{-|<knobjconffile>}] [-o{-|<outfile>}]
```

In the such synopsis form the *ufill* reads such data from standard input, fills Ntuple(s)/histogram(s) as described in default configuration files, and writes it(s) to HBOOK RZ file `<run_name>`*.paw* or ROOT file `<run_name>`*.root* for further analysis by PAW [4] or ROOT [5], respectively (PAW or ROOT mode defined at compile time).

At startup *ufill* reads configuration files in the ***RUN.conf(5)*** (see 2.3), ***cell.conf(5)*** (see 2.1), and ***knobj.conf(5)*** (see 2.2.3) formats; initializes structures `pdat`, `cell`, `knvar`, `knfun`, `knobj` (see *pdata.h*, 2.1, 2.2), performs create loop over all initialized `knobj`s and generates `PROG_BEG` event. After that it reads experimental data stream from standard input and for each obtained event increments the global counter, corresponding to type of this event, and performs calculation loop over all initialized `cells` and fill/clean loop over all initialized `knobj`s. At data stream `EOF` state obtaining *ufill* generates `PROG_END` event.

At `PROG_BEGIN` and `PROG_END` events also performed calculation loop over all initialized `cells` and fill/clean loop over all initialized `knobj`s.

The default behavior of *ufill* may be changed by following options:

`-c<runconffile>` Use `<runconffile>` as SPHERE experimental data configuration file (see 2.3). `-c-` means use compiled–in default for `<runconffile>` (constructed from `<run_name>` by appending "`.conf`" extension).

`-s<cellconffile>` Use `<cellconffile>` as configuration file for universal data containers, cells (see 2.1). `-s-` means use compiled–in default for `<cellconffile>` (constructed from `<program_name>`[4] by prepending "c" and appending "`.conf`").

`-k<knobjconffile>` Use `<knobjconffile>` as configuration file for universal presenter objects, knobjs (see 2.2.3). `-k-` means use compiled–in default for `<knobjconffile>` (constructed from `<program_name>` by prepending "p" and appending "`.conf`").

`-o<outfile>` Use `<outfile>` as name of RZ/ROOT file, to which filled Ntuple(s) will be written.
"`.paw`"/"`.root`" extensions will be added correspondingly. `-o-` means use compiled–in default for `<outfile>` (constructed from `<run_name>` by appending proper extension).

The *ufill* utility exits 0 on success, and > 0 on error.

2.4.2 *calibtof(1)* utility

calibtof(1) utility destined for calibrating of time–of–flight (TOF) information, contained in SPHERE experimental datafile(s).

```
calibtof [-q] [-f<listfile>] [-n<basename>] [-p[<dirname>]] [-w]
        [-b<Base>] {d|p|k|pi} <Mom> [<Ll> <Lh>]
```

[4]`<program_name>` need to be substituted by name, under which this ***ufill(1)*** utility was compiled (usually "*ufill*")

In the such synopsis form the *calibtof* reads list of such datafile's name(s) from the file *files.lst* (one full filename per string), TOF initial calibration information from the file *calibtof.ini* in ***tof(5)*** format (in particular, length of TOF baseline `<Base>`); after that *calibtof* reads each datafile, calculates new calibration information for deuterons, protons, kaons, or pions with momentum `<Mom>`, and writes it in files *calibtof.out*, and *calibtof.cal*. Lower `<Ll>` and upper `<Lh>` limits for the "window" in the $(F561\_TDC+F562\_TDC)/2$ spectrum are equals by default 0 and 1100 channels, correspondingly, if not specified in the command string. Such "window" need to separate particles of type, by which calibration will be performed.

For more details see ***calibtof(1)***, ***tof(3)***, ***tof(5)***.

2.4.3 *ckdata(1)* utility

ckdata(1) utility destined for SPHERE experimental data consistency check.

```
ckdata [-c{-|<runconffile>}] [-s{-|<cellconffile>}]
          [-k{-|<knobjconffile>}]
```

In the such synopsis form the *ckdata* reads such data from standard input and checks it in accordance with default configuration files. If `<knobjconffile>` does not contains any call of "`ck`" functions, only data format (not contents) are checked (by ***get_data(3)*** built–in checks). If `<knobjconffile>` contains call(s) of "`dump`" functions, *ckdata* dumps (writes in ASCII representation) corresponding data contents to standard error output.

The default behavior of the *ckdata* may be changed by some options (see *ufill*'s options description above).

The *ckdata* exits 0 on success, and > 0 on error.

3 CAMAC configuration scheme

A CAMAC hardware representation scheme is designed to formalize and automatize C coding of the software pieces deals with CAMAC (in particular CAMAC interrupt handler (see [2]) and some DAQ utilities) for expansive CAMAC setups.

Basic idea is to provide:

1. "geography" point of view to CAMAC hardware – some kind of configuration database, contains information about physical place of each CAMAC hardware module, used during a some RUN. Let to name it as a global CAMAC description array.
2. some kind of "library", contains C code pieces, implements for each kind of used CAMAC hardware modules (at least) following:
   - real reading data from CAMAC;
   - testing corresponding CAMAC module;
   - data obtaining simulation without real reading (for debug purposes).
3. "event" point of view to CAMAC hardware – experimental data representation in terms of used CAMAC hardware modules for each event type, produced by setup.

4. utilities for CAMAC C code generation and CAMAC testing, based on previous items.
5. code for control over whole DAQ system, RUN independent part (f.e. event type recognition, start, stop, triggers enable/disable operations, etc.).
6. RUN dependent part of previous item (f.e. initialization, clean CAMAC operations, etc.).

Here we describe a RUN independent part of such idea implementation (items 2, 4, and 5 above).

3.1 CAMAC hardware library

Item 2 is implemented in the *hwconf.c* and *hwconf.h* files. *hwconf.h* declares (at least) the following things:

```
typedef int (*f_read)(FILE *, char *, int *, u_short, int, int);
typedef int (*f_test)(int, int);
typedef int (*f_gen)(FILE *, char *, int *, u_short);
struct station {
        char used;
        enum hw_keys kf;
        f_read fr;
        f_test ft;
        f_gen fg;
};
struct crate {
        char used;
#define CAMAC_MAX_N     23      /* last valid station N (from 1) */
        struct station sts[CAMAC_MAX_N+1];
};
```

used for construction of the global CAMAC description array.

hwconf.c implements the following things:

```
#include <hwconf.h>
int hw_search(FILE *stream, char *buf, enum hw_keys key,
        int *offset, u_short type, int flag)
int r_<module_name>(FILE *stream, char *buf, int *offset,
        u_short type, int cr, int st)
int t_<module_name>(int cr, int st)
int g_<module_name>(FILE *stream, char *buf, int *offset,
        u_short type)
inline void select_crate(FILE *stream, int cr)
```

`hw_search()` searches for CAMAC crate.station combination in the CAMAC description array `crs[]` by key `hw_keys key` and executes function, corresponding to `flag` (valid values `#define`'d in the *hwconf.h* header file), with `stream`, `buf`, offset, type parameters. Search will either traverse entire CAMAC hardware structure, if flag == HW_TEST or flag == HW_INIT, or terminate after first match occur, if flag == HW_READ or flag == HW_GEN.

r_<module_name>()[5] generates C code for dealing with CAMAC module <module_name>, situated in the crate cr, station st, for event of type type production, writes such code into stream stream, and increments *offset (in sizeof(u_short) units) in accordance with amount of data, should be read from CAMAC. The *buf points to string, used as name of the event storage in produced C code.

t_<module_name>() tests CAMAC module <module_name>, situated in the crate cr, station st.

g_<module_name>() works like r_<module_name>(), but produced C code only fills the piece of event storage by (some) arbitrary data instead of really read data from CAMAC. Useful for event generator writing.

select_crate() writes C code for CAMAC crate cr selection into stream stream.

All functions returns 0 on success, and nonzero otherwise.

3.2 Command interface to CAMAC configuration scheme

Item 4 is implemented by the ***gen_gr(1)*** and ***ctest(1)*** utilities.

The *gen_gr* – C code generator for CAMAC kernel modules of the SPHERE DAQ system.

```
gen_gr [-i] [-g] <event_kind>
```

In the such synopsis form the *gen_gr* generates C code (in the #define form by default) for reading event <event_kind> from CAMAC in accordance with compiled–in hardware description (see discussion about *hwconf.c* above), and writes it to standard output.

The default behavior of the *gen_gr* may be changed by following options:

-i Generate C code in the form of inline void function instead of #define by default.

-g Generate C code for event generator instead of CAMAC reading by default.

The *gen_gr* exits 0 on success, and >0 on error.

The *ctest* – CAMAC test utility for the SPHERE DAQ system.

```
ctest
```

In the such synopsis form the *ctest* performs CAMAC tests in accordance with compiled–in hardware description (see discussion about *hwconf.c* above).

The *ctest* exits 0 on success, and >0 on error.

3.3 RUN independent DAQ control code

Item 5 is implemented in the *hardware.h* file, contains a generic CAMAC hardware macro definitions. In the current implementation these are:

[5]<module_name> need to be substituted by name of a CAMAC hardware module (for example, “4zcp397” – some ADC, “4vcp369” – some TDC, “2sc417” – some scaler, etc.)

- `DAQ_START()` starts of data acquisition.
- `DAQ_STOP()` stops of data acquisition.
- `TRIG_START()` enables triggers.
- `TRIG_STOP()` disables triggers.
- `DEADTIME_STOP()` disables dead time.
- `CHECK_INTR(int *res)` checks interrupt validity, and fills `res` by 0 on success, > 0 on failure.
- `IREG_READ(u_short *ireg)` performs input/output register reading, and fills `ireg` by obtained value.
- `EVENT_TYPE(u_short ireg, u_short *h)` by supplied `ireg` value recognizes event type, calculates corresponding to it offset in `type[NEVTYPES]` array, and fills `h` by such offset value. Valid offsets are 0–`(NEVTYPES-1)`.
- `CAMAC_INTR_ON()` enables CAMAC interrupt generation.
- `CAMAC_INTR_OFF()` disables CAMAC interrupt generation.
- `CAMAC_INTR_RESET(int *res)` checks CAMAC interrupt presence and resets it. Returns 0 on absence, > 0 on presence.

bRUN.h selects correct RUN dependent *b*`<run_name>`*.h* header file by `#ifdef` mechanism.

hRUN.h selects correct RUN dependent `<run_name>`*hard.h* header file by `#ifdef` mechanism.

4 Software dependencies and portability

The SPHERE *offline* system currently uses the following third–party software:
cernlib package – HBOOK `Ntuples` support for the ***ufill(1)***[6].
ROOT package – ROOT `TNtuples` support for the ***ufill(1)***[6].

All software packages mentioned above are free distributable, so does not limits a portability of the *offline* system.

All *offline* code written on C and C++ for more or less generic modern UNIX–like environment. All code for user context processes expected to be easy portable to OS's other than FreeBSD.

Makefiles follows the BSD make's (PMake) syntax and uses BSD system makefiles *bsd.prog.mk*, *bsd.lib.mk*, *bsd.man.mk*, and *bsd.subdir.mk* from the */usr/share/mk* . The upper directory of the *offline* system source distribution contains *Makefile.def*, which initializes variables, used at *offline* system making, to default values. It `.include`'d by each Makefile in the *offline* source tree.

In the current implementation the *offline* system based on FreeBSD operating system (4.x version at present time). *offline* sources are distributed as archive
`offline-<yyyymmdd>.tar.gz`[7] (please contact `isupov@moonhe.jinr.ru`).

[6]***ufill(1)*** can be based on the *cernlib* package OR on the *ROOT* package.
[7]<yyyymmdd> means some actual year, month, and day numbers.

Acknowledgements

Author have a pleasure to thank ~~to~~ S.G.Reznikov and K.I.Gritsaj for useful discussions, and ~~to~~ A.Yu.Semenov for submitting of the ***calibtof(1)***'s initial version.

References

[1] S.V.Afanasiev et al. **Measurement of the tensor analyzing power A_{yy} in inclusive breakup of 9 GeV/c deuterons on carbon at large transverse momenta of protons.** Physics Letters B 434, 1998, p.21.

[2] Gritsaj K.I., Isupov A.Yu. **A trial of distributed portable data acquisition and processing system implementation: the *qdpb* – Data Processing with Branchpoints.** Communication of JINR E10–2001–116, Dubna, 2001.

[3] Brun R., Lienart D. **HBOOK Users Guide.** CERN Program Library Entry Y250. CERN, Geneva, Switzerland, 1987.

[4] Brun R., Couet O., Vandoni C., Zanarini P. **PAW. Physics Analysis Workstation.** CERN Program Library Entry Q121. CERN, Geneva, Switzerland, 1990.

[5] Brun R., Buncic N., Fine V., Rademakers F. **ROOT. Overview.** CodeCERN, 1996.

Received by Publishing Department
on September 26, 2001.

Исупов А.Ю. E10-2001-199
Конфигурируемые представления данных
и аппаратуры КАМАК для реализации систем сбора
и обработки данных установки СФЕРА

Описывается реализация конфигурируемого представления экспериментальных данных для использования в системах сбора и обработки данных установки СФЕРА (ЛВЭ ОИЯИ). Излагается программная схема конфигурируемого описания аппаратуры КАМАК установки СФЕРА, предназначенная для реализации сбора данных на основе системы *qdpb*.

Работа выполнена в Лаборатории высоких энергий ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна, 2001

Isupov A.Yu. E10-2001-199
Configurable Data and CAMAC Hardware Representations
for Implementation of the SPHERE DAQ and Offline Systems

An implementation of the experimental data configurable representation for using in the DAQ and offline systems of the SPHERE setup at the LHE, JINR is described. A software scheme of the SPHERE CAMAC hardware's configurable description, intended to online data acquisition (DAQ) implementation based on the *qdpb* system, is issued.

The investigation has been performed at the Laboratory of High Energies, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna, 2001

Макет Т.Е.Попеко

Подписано в печать 11.10.2001
Формат 60 × 90/16. Офсетная печать. Уч.-изд. л. 1,9
Тираж 300. Заказ 52896. Цена 1 р. 30 к.

Издательский отдел Объединенного института ядерных исследований
Дубна Московской области