E10-2003-187

A. Yu. Isupov

SPHERE DAQ AND OFF-LINE SYSTEMS: IMPLEMENTATION BASED ON THE *qdpb* SYSTEM

1 Introduction

Data acquisition system's implementation for the SPHERE experiment [1] based on the distributed portable data acquisition and processing system *qdpb* [2] as well as on the configurable experimental data and CAMAC hardware representations [6] now. This paper describes implementation of SPHERE DAQ and offline code, which depends on hardware and data layouts and, therefore, need to be varied from RUN[1] to RUN.

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 4.2.1. Subjects of substitution by actual values are enclosed in the angle brackets: `<run_name>`. All mentioned trademarks are properties of its respective owners.

2 SPHERE CAMAC hardware subsystem

Online electronics of the SPHERE setup [1] based on a CAMAC hardware currently, so hardware subsystem in terms used by the *qdpb* system [2] is a CAMAC subsystem only.

2.1 RUN dependent CAMAC code

General idea about CAMAC configuration is formulated and RUN independent code part is explained in [6]. Here we describe a RUN dependent part of such idea implementation:

1. "geography" point of view to CAMAC hardware (so called global CAMAC description array).
2. "event" point of view to CAMAC hardware.
3. code for control over whole DAQ system, RUN dependent part.

A global CAMAC description array (item 1) and experimental data representation in terms of CAMAC hardware modules (item 2) are implemented in the *b*`<run_name>`.*[2] family of files (one *.c* C source and one *.h* C header per RUN).

b`<run_name>`*.h* defines a CAMAC hardware keys – symbolic names for CAMAC crate.station combinations – in the enumerate form:

```
enum hw_keys { NoField, <other field names here>, <...> };
```

b`<run_name>`*.c* contains:

- The global CAMAC description array definition and initialization:

```
const struct crate crs[<crates_in_branch>] = { <...> };
```

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

[2] `<run_name>` anywhere through this report need to be substituted by name of an experimental RUN (for example, "nov96", "jun97", "jun98", etc.)

where <crates_in_branch> need to be substituted by correct value, #define'd in the *hwconf.h* header file.

- gr_<event_kind>() functions implementation.

```
#include <b<run_name>.h>
int gr_<event_kind>(FILE *stream, char *buf, int flag)
```

These routines destined to represent the binary experimental data layout in terms of CAMAC crate.station combinations, from which their are read.

gr_<event_kind>() generates C code, which produce <event_kind> event, and outputs it into stream. buf is a string, contains name for event storage in produced C code. If flag == HW_READ (#define'd in the *hwconf.h* header file), this code really deals with CAMAC, if flag == HW_GEN, this code only fills storage by (some) values.

The gr_<event_kind>() returns 0 on success, and positive wrong offset value (in the sizeof(u_short) units) on failure.

RUN dependent DAQ control code (item 3) is implemented in <run_name>*hard.h* files family (one per RUN), which #includes <run_name>_<event_kind>.*h*, <run_name>_*camac_sc_reset.h*, and <run_name>_*camac_init.h* files, generated by the ***gen_gr(1)*** (see [6]). So, RUN dependent DAQ control code contains (in the current implementation) the following CAMAC hardware macro definitions:

- CAMAC_INITIALIZE(int *res) performs CAMAC initialization, and fills res by 0 on success, > 0 on failure.
- CAMAC_FINISH(int *res) performs CAMAC finish operations, and fills res by 0 on success, > 0 on failure.
- CAMAC_CYC_BEG() deals with CAMAC and produces DATA_CYC_BEG event.
- CAMAC_DAT_0() deals with CAMAC and produces DATA_DAT_0 event.
- CAMAC_CYC_END() deals with CAMAC and produces DATA_CYC_END event.
- CAMAC_DAT_N(u_short h) produces DATA_DAT_<N> event of type type[h].
- CAMAC_CLEAN() cleans CAMAC at errors in interrupt handler.
- CAMAC_SCALERS_RESET() performs non-blocked scalers reset.

2.2 SPHERE CAMAC interrupt handler implementation

User handler of CAMAC interrupts for SPHERE setup is implemented by:

- CAMAC configuration scheme, described in [6] (see also 2.1)
- KK009/KK012 memory access macros (see [3]) for CAMAC handling,
- kernel context interface to packet handling (see [2]),
- kernel context interface to branch point (see [2]).

In the current implementation it named hand and loaded into kernel as follows:
kldload hand_mod.ko

For actual version of such handler source see *camac_modules/handler/hand.c* file in the *qdpb* system [2] distribution tree.

handconf – configuration utility for hand CAMAC module of the SPHERE DAQ system.

`handconf [-v] [-d{<driver>|-}] <module>`
`handconf [-v] -t <module>`

In the first synopsis form the *handconf* utility configures the specified module `<module>` for work with driver "kk0" by default.

In the second synopsis form the *handconf* utility tests configuration of the specified module `<module>` and writes it to standard error output.

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

`-d<driver>` Configure module for work with driver `<driver>` instead of default "kk0". `-d-` means use compiled–in default for `<driver>`. Default driver name may be changed at *handconf* compile time.

`-v` Produce verbose output instead of short by default.

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

`handoper` – control utility for hand CAMAC module of the SPHERE DAQ system.

`handoper [-v] [-b<#>] start|stop|status|init|finish|quecl|cntcl`

In the such synopsis form the *handoper* performs `oper()` call with sub–function `fun`, defined by first supplied argument, on the CAMAC module (see also [3] for more details) attached to the 0^{th} branch, and writes report about that action to the standard error output. The *handoper* may be used, for example, for implementation of some actions in the ***sv.conf(5)*** file (see also 4.4 and supervisor software module description in [2]).

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

`-v` Produce verbose output instead of short by default. With this option *handoper* also uses `oper()` call with `HANDGETSTAT fun` (see also [3] for more details).

`-b<#>` Deal with module, attached to the `#`th branch, instead of 0^{th} by default.

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

3 RUN dependent *offline* code

All code, dependent on experimental data contents and layout, and therefore tends to be changed between different accelerator RUNs (denoted below as `<run_name>`) is grouped together by so called *offline* system, which provides set of libraries *lib*`<run_name>` – one library per accelerator RUN.

Each *lib*`<run_name>` contains all `<run_name>` specific routines (see 3.1) with corresponding RUN independent low level support (see *bytemacros.h* header file), and relatively RUN independent high level routines:

- datafile(s) handling (see 3.2) on the ***packet(3)***'s interface (see [2]) base,
- routines for packet (event) types handling (see 3.3), etc.

3.1 Data contents description

`#define <run_name>`
`#include <eRUN.h>`

The *eRUN.h* contains `#include`'s of all exists *e*`<run_name>`*.h* files, each of which contains description of the experimental data layout in the corresponding RUN `<run_name>`.

The *fRUN.h* header file declares and the *f*`<run_name>`*.c* source file implements routines, destined to convert from the binary experimental data to corresponding data structures and vise versa.

```
#define <run_name>
#include <fRUN.h>
void b2s_<event_kind>(u_char *buf, char *data)
void s2p_<event_kind>(char *data, pdata *pdat)
void s2b_<event_kind>(u_char *buf, char *data)
```

`b2s_<event_kind>()` fills already allocated `data_<event_kind>` structure, pointed by `*data`, from `*buf`, contains binary representation of event of kind `<event_kind>`. So, `b2s_<event_kind>()` performs conversion from binary to structure experimental data representation, after which user can retrieve event fields by its names instead of its offsets.

`s2p_<event_kind>()` directs `ptr` members of already allocated array of `pdata` structures `*pdat` (see 3.2) into corresponding fields in the `data_<event_kind>` structure, pointed by `*data`. After that user can retrieve event fields by its index numbers in the `*pdat` array instead of its names.

`s2b_<event_kind>()` fills already allocated buffer `*buf` from the `data_<event_kind>` structure, pointed by `*data`. So, `s2b_<event_kind>()` performs conversion from structure to binary experimental data representation. These functions destined primarily for data makers (usually elements of hardware subsystem) purposes.

3.2 High–level data reading

```
#define <run_name>
#include <eRUN.h>
#include <get_data.h>

int get_pack(FILE *finput, char **data, char **data_ptr,
        int (**usr_func)(char *, char *, int), int flag)
int get_burst(FILE *finput, char **data_buf, u_long *counters_buf)
#define get_data(in, dat, dat_p, usr_fn, flag)      \
        get_pack((in), (dat), (dat_p), (usr_fn), (flag))
```

`get_pack` and `get_burst` routines are destined to read a binary experimental data in the ***packet(3)*** format of the *qdpb* system [2] and to present it as corresponding data structures for more comfortable using.

`get_pack()` organize loop for read whole packet datafile(s). Data read from stream, pointed by `*finput`, if it is not `NULL`, otherwise data read from branch point by its user context interface (see [2]). The `**data` must contain pointers to data structures `data_<event_kind>` (defined in the *e*`<run_name>`*.h* header file, see 3.1), which will be filled by read data. The `**data_ptr` may contain pointers to arrays of `pdata` structures, defined in the *pdata.h* header file as follows[3]:

[3]String representations for possible variable types produced by simple doublequoting of presented preprocessor variables "`type_*`".

```
typedef struct {
#define TVNAME_LEN      10
        char tvar[TVNAME_LEN]; /* name for value */
        u_char flag;           /* variable handled if (flag) */
/* u_char flag may be: */
#define F_NULL          0      /* variable not handled at all */
#define F_HANDLE        1      /* variable will be handled */
        u_char type;           /* value type */
/* u_char type may be: */
#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
        void *ptr;             /* pointer to value itself */
} pdata;
```

or array of NULL pointers, if you need not pdata interface. The array of pointers to functions (**usr_func)() must contain pointers to user data handling procedures (or NULL), which will (will not) be called at each occurrence of corresponding event kind of experimental data. Up to NEVTYPES handlers for working with each of data_<event_kind> structure may exists. Handlers must be declared as int <sm_name><kind>(char *ch, char *ch1, int flag)[4] and returns nonzero at error. Pointer *ch must be used into handler as pointer to the corresponding data structure, pointer *ch1 optionally may be used as pointer to array of structures pdata, flag may control the handlers behavior (for example, it may indicate main or secondary data stream).

get_burst() gets whole burst from packet datafile, assuming, that first packet of burst has DATA_CYC_BEG type (#define'd in the *pack_types.h*), last packet of burst has DATA_CYC_END type, and second, ..., previous of last packets are of other DATA_DAT types. Data read from stream, pointed by *finput, if it is not NULL, otherwise data read from branch point by its user context interface (see [2]). The **data_buf must contain pointers to already allocated at least NEVTYPES arrays of data structures data_<event_kind> (defined in the *e*<run_name>*.h* header file, see 3.1), which will be filled by read data. The *counters_buf must contain pointer to already allocated array, contains at least NEVTYPES members, for storing number of events of each types, encountered during burst read. So, loop over bursts organization and "event handlers"

[4] <sm_name> need to be substituted by some C identifier name,
<kind> – by short name of event kind (for example, "dc0", "dd0", "dc1", etc.)

calling is a user responsibility, but (s)he has whole current burst (instead of current event in the `get_pack()` scheme) available to analysis at each time moment.

Macro `get_data()` organize loop for read whole datafile(s) in the current format, defined by `GETPACK`, and has interface the same as `get_pack()`.

`get_pack()` and `get_data()` returns 0 on success, positive values if error occurred in user handlers, and negative values if error occurred while data reading or in these functions itself.

The `get_burst()` returns 1 on success, 0 if end–of–data occurred and < 0 if error.

The following error codes may be set in `errno`:

`[EINVAL]` Invalid argument(s) supplied to `get_pack()` or `get_burst()`.

`[ERANGE]` Data, read by `get_pack()`, are damaged (invalid chess codes, etc.)

`[EIO]` I/O error occurred.

`[EMSGSIZE]` Unexpected type of event read by `get_burst()`.

3.3 Packet (event) classification

```
#include <pack_types.h>
```

The *pack_types.h* header file must contains predefined packet types, classes, and kinds (in order from more general to more particular). In the current implementation value of `u_short packet.header.type` (see [2]), used for packet kind representation in packet header, divided into set of subranges as follows:

```
/* Predefined packet types */
#define DATA_PACK       0
#define DATA_RANGE      9999
/* Predefined packet classes in DATA_PACK type */
#define DATA_DAT        DATA_PACK
#define DATA_DAT_RANGE  999
#define DATA_CYC        (DATA_DAT+DATA_DAT_RANGE+1)
#define DATA_CYC_RANGE  999
/* Predefined packet kinds in DATA_DAT class */
#define DATA_DAT_0      DATA_DAT
#define DATA_DAT_1      (DATA_DAT+1)
...
/* Predefined packet kinds in DATA_CYC class */
#define DATA_CYC_BEG    DATA_CYC
#define DATA_CYC_END    (DATA_CYC+1)
```

`offbystr()`, `offbyevtype()`, `evtypebyoff()`, `evtypebystr()`, `strbyevtype()` – routines intended to packet (event) types handling.

```
#define <run_name>
#include <pack_types.h>
u_short offbystr(char *tok)
u_short offbyevtype(u_short ptype)
u_short evtypebyoff(u_short offset)
```

`u_short evtypebystr(char *tok)`
`char *strbyevtype(u_short ptype, char *dst)`

`offbystr()` returns offset in `<array>[NEVTYPES]`[5] arrays, defined in the *e*`<run_name>`*.h* header file (see 3.1), for packet type, represented by name `*tok`.

`offbyevtype()` returns offset in `<array>[NEVTYPES]` arrays for packet type, represented by value `ptype` (`packet.header.type`).

`evtypebyoff()` returns packet type value for offset `offset` in `<array>[NEVTYPES]`.

Macro `evtypebystr()` returns `ptype` for packet type, represented by name `*tok`.

Macro `strbyevtype()` fills string representation in the already allocated destination `dst` of at least `PTYPE_STR_MAX` (`#define`'d in the *pack_types.h* header file) size for packet type, represented by value `ptype`, and returns pointer to filled string.

`offbystr()` returns `(u_short)(-1)` if type name submitted is unknown.

`offbyevtype()` returns `(u_short)(-1)` if type value submitted is unknown.

`evtypebyoff()` returns `(u_short)(-1)` if `offset >= NEVTYPES`.

`evtypebystr()` returns `(u_short)(-1)` if type name submitted is unknown or `offset >= NEVTYPES`.

`strbyevtype()` returns NULL if type value submitted is unknown or `offset >= NEVTYPES`.

4 SPHERE DAQ specific software modules

SPHERE DAQ system (in single host configuration) uses at least ***writer(1)*** work module, ***bpget(1)*** service module, and (optionally) supervisor ***sv(1)*** control module (see [2]) from generic software modules assortment provided by the *qdpb* system. Here we discuss only SPHERE specific software modules.

4.1 Work module: statistic collector *statman*

The statistic collector *statman* implemented as work module of packet stream terminator type (in terms of the *qdpb* system [2]). It supports in the shared memory region(s) some data objects (in particular histograms and counters), used by data presenter module(s) (see 4.2.1, 4.2.2).

`statman [-o] [-b<bpstat>] [-c{-|<runcffile>}] [-s{-|<cellcffile>}]`
`[-k{-|<knobjcffile>}] [-i{-|<cleancffile>}] [-p{-|<pidfile>}]`

In the such synopsis form the *statman* reads packets from standard input, collects information from each `packet.data` in accordance with default configuration files, and stores it in the shared memory.

At startup *statman* reads configuration files in ***RUN.conf(5)*** (see [6]), ***cell.conf(5)***, ***knobj.conf(5)*** and ***clean.conf(5)*** (see below and [6]) formats; initializes structures `pdat`, `cell`, `knvar`, `knfun`, `knobj` (see 3.2, [6]), performs create loop over all initialized `knobj`s and generates `PROG_BEG` event. After that it reads packet stream from standard input and for each obtained packet increments the global counter, corresponding to type of this packet, and performs calculation loop over all initialized `cell`s

[5] `<array>` need to be substituted by one of the `max_memb`, `len`, `type`, `chan_max`, `k_shmid`, `k_semid` values

and fill/clean loop over all initialized knobjs. At packet stream EOF state obtaining or SIGTERM signal catching statistic collector generates PROG_END event, so using SIGKILL signal for statistic collector termination not recommended.

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

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

-o Don't remove shared memory at exit (useful for offline purposes).

-b<bpstat> [6] Use branch point (see [2]) as input instead of standard input, and open it at <bpstat> state. (Available only for systems, where branch point is implemented).

-p<pidfile> At startup write own process identifier (PID) in <pidfile>. -p- means use compiled–in default for <pidfile>.

-c<runcffile> Use <runcffile> as SPHERE experimental data configuration file (see [6]). -c- means use compiled–in default for <runcffile> (constructed from <run_name> by appending ".conf" extension).

-s<cellcffile> Use <cellcffile> as configuration file for universal data containers, cells (see [6]). -s- means use compiled–in default for <cellcffile> (constructed from <program_name>[7] by prepending "c" and appending ".conf").

-k<knobjcffile> Use <knobjcffile> as configuration file for universal presenter objects, knobjs (see [6]). -k- means use compiled–in default for <knobjcffile> (constructed from <program_name> by prepending "p" and appending ".conf").

-i<cleancffile> Use <cleancffile> as cleaning list for knobjs (see [6]). -i- means use compiled–in default for <cleancffile> (constructed from <program_name> by prepending "i" and appending ".conf").

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

The *statman* ignores SIGQUIT signal. The SIGHUP signal causes to reread configuration files <runcffile>, <cellcffile>, and <knobjcffile> (with the same names, as used at startup) which leads to whole statistics and cell results cleaning (so this equivalent to startup initialization). The SIGINT signal causes to reread configuration file <cellcffile> (with the same name, as used at startup) but without cell results cleaning (so this can be used to change some cell "programs" without results destroy). For user termination in accuracy manner SIGTERM signal must be used. The SIGUSR1 signal cleans whole collected statistics (include internal event counters), the SIGUSR2 signal cleans it in accordance with contents of file <cleancffile>. Both SIGUSR1 and SIGUSR2 are cleans results of all cells.

The *statman*'s configuration file in the ***knobj.conf(5)*** format (see [6]) can contains only declarations of known objects with types, supported by *statman*. Such knobj types currently are "hist", "hist2" (maintains histogram array in the shared memory),

[6] <bpstat> need to be substituted by r for "run", s for "stop", and d for "discard"

[7] <program_name> need to be substituted by name, under which statistic collector was compiled (usually "*statman*")

"cnt" (maintains counters array in the shared memory), "coord", and "coord2" (maintains coordinate "histogram" array in the shared memory).

For such known object's declaration entries in the ***knobj.conf(5)*** format the first (name), third (type), fifth (fill dependence), sixth (fill condition), and seventh (clean dependence) fields has its standard in such format meanings.

Second (creator parameters), fourth (filler parameters), eighth (cleaner parameters) and ninth (remover parameters) fields need to complain program interface, provided by "hist", "hist2", "cnt", "coord" and "coord2" known functions families.

For example, declaration of "hist" "hist2", "cnt", "coord" and "coord2" known objects:

```
Obj0013   13;1025;type_ULong;create;shmid;semid   hist \
   tdc0;type_UShort;13;semid;type_ULong   DATA_DAT_0   - \
   NEVERMORE   1025;type_ULong;13;semid   13;shmid;semid
Obj0033 33;1025;1025;type_ULong;create;shmid;semid hist2 \
   tdc0;tdc1;type_UShort;33;semid;type_ULong;1025 \
   DATA_DAT_0 - NEVERMORE 1025;1025;type_ULong;33;semid \
   33;shmid;semid
Obj0041   41;5;create;shmid;semid;type_ULong   cnt \
   41;3;semid;reset;type_ULong;cnt21;cnt22;cnt23 DATA_CYC_END   - \
   NEVERMORE  41;5;semid;type_ULong  41;shmid;semid
Obj0008   8;16;type_ULong;create;shmid;semid   coord \
   hr0;type_UShort;8;semid;16;add;type_ULong  DATA_DAT_0  - \
   NEVERMORE  16;type_ULong;8;semid  8;shmid;semid
Obj0028   28;16;16;type_ULong;create;shmid;semid   coord2 \
   hr0;hr1;type_UShort;28;semid;16;16;add;type_ULong \
   DATA_DAT_0 - NEVERMORE 16;16;type_ULong;28;semid \
   28;shmid;semid
```

To simplify writing of such configuration files the ***gen_prescfg(1)*** utility is proposed (see 4.3).

For example, following prototype entries leads to generation of the known object declarations presented above:

```
hist 13 1 -1 1025 ULong create shmid semid tdc%0N UShort DAT_0 - N
hist2 33 1 -1 1025 1025 ULong create shmid semid tdc%0.1n adc%3.2n \
     UShort DAT_0 - N
cnt 41 1 -1 3 create shmid semid ULong reset cnt%21N CYC_END - N
coord 8 1 -1 16 ULong create shmid semid hr%0N UShort add DAT_0 - N
coord2 8 5 -1 16 16 ULong create shmid semid hr%0.2n HR%1.2n \
     UShort add DAT_0 - N
```

4.2 SPHERE DAQ control modules

In the current implementation of the SPHERE DAQ the following control modules (in terms of the *qdpb* system [2]) are provided:

- the histograms data presenter ***histview(1)***,
- the counters data presenter ***cntview(1)***,

4.2.1 Histograms data presenter

The histograms data presenter *histview* is a graphical statistic representation utility for the SPHERE DAQ system. It reads data objects, supported by statistic collector (see 4.1), and builds for it some human readable graphical representation.

`histview [-k{-|<knobjconffile>}] [-p{-|<pidfile>}] [-t<sleeptime>]`

In the such synopsis form the *histview* reads statistic, collected in the shared memory by ***statman(1)***, interprets it in accordance with default configuration file in the ***knobj.conf(5)*** (see below) format, and produces graphics representation of such statistic (in the current implementation – using ROOT [4] facilities).

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

`-p<pidfile>` At startup write own process identifier (PID) in `<pidfile>`. `-p-` means use compiled–in default for `<pidfile>`.

`-k<knobjconffile>` Use `<knobjconffile>` as configuration file for universal presenter objects, knobjs (see [6]). `-k-` means use compiled–in default for `<knobjconffile>` (constructed from `<program_name>` by prepending "p" and appending "`.conf`").

`-t<sleeptime>` Use `<sleeptime>` (in seconds) as time interval between two dumps, instead of use compiled–in default 5 seconds.

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

The *histview* ignores `SIGINT` and `SIGQUIT` signals. The `SIGHUP` signal causes to reread configuration file (with the same name, as used at startup). For user termination in accuracy manner the `SIGTERM` signal must be used. The `SIGUSR1` signal stops statistic reading and representing, the `SIGUSR2` continues one.

The *histview*'s configuration file in the ***knobj.conf(5)*** format (see [6]) can contains only declarations of known objects with types, supported by *histview*. Such knobj types currently are "`TH1`" and "`TH2`" (ROOT [4] based 1–D and 2–D histograming and graphic presentation of histogram array taked from shared memory).

For such known object's declaration entries in the ***knobj.conf(5)*** format the first (`name`), third (`type`), fifth (fill dependence), sixth (fill condition), and seventh (clean dependence) fields has its standard in such format meanings.

Second (creator parameters), fourth (filler parameters), eighth (cleaner parameters) and ninth (remover parameters) fields need to complain program interface, provided by "`TH1`" and "`TH2`" known functions families.

For example, declaration of "`TH1`" and "`TH2`" known objects:

```
Obj0009 hr1;HR1;16;0;16;TH1F;9;shmid;semid;shmem;canvas;9 \
   TH1   type_ULong;16;0;16;TH1F;9;semid;shmem;canvas;9 \
   DATA_DAT_0   -   NEVERMORE   TH1F;canvas;9   TH1F;canvas;9
Obj0013 tdc0;TDC0;1025;0;1025;TH1F;13;shmid;semid;shmem;canvas;13 \
   TH1   type_ULong;1025;0;1025;TH1F;13;semid;shmem;canvas;13 \
   DATA_DAT_0   -   NEVERMORE   TH1F;canvas;13   TH1F;canvas;13
Obj0029 hx_hy;HX_HY;16;0;16;16;0;16;TH2F;29;shmid;semid;shmem;canvas;29 \
   TH2 type_UShort;16;0;16;16;0;16;TH2F;29;semid;shmem;canvas;29;16;LEGO \
   DATA_DAT_0   -   NEVERMORE   TH2F;canvas;29   TH2F;canvas;29
```

The ***gen_prescfg(1)*** (see 4.3) generates known object declarations, presented above, from the following prototype entries:

```
TH1 9 1 hr%1N HR%1N 16 0 16 TH1F -1 shmid semid shmem
     canvas -1 ULong DAT_0 - N
TH1 13 1 tdc%0N TDC%0N 1025 0 1025 TH1F -1 shmid semid shmem
     canvas -1 ULong DAT_0 - N
TH2 29 1 hx_hy HX_HY 16 0 16 16 0 16 TH2F -1
     shmid semid shmem canvas -1 ULong DAT_0 - N 16 LEGO
```

4.2.2 Counters data presenter

The counters data presenter *cntview* is a textual statistic representation utility for the SPHERE DAQ system. It reads data objects, supported by statistic collector (see 4.1), and builds for it some human readable textual representation.

`cntview [-k{-|<knobjconffile>}] [-p{-|<pidfile>}] [-t<sleeptime>]`

In the such synopsis form, the *cntview* reads statistic, collected in the shared memory by ***statman(1)***, interprets it in accordance with default configuration file in the ***knobj.conf(5)*** (see below) format, and dumps (writes in ASCII representation) such statistic to standard error output.

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

`-p<pidfile>` At startup write own process identifier (PID) in `<pidfile>`. `-p-` means use compiled–in default for `<pidfile>`.

`-k<knobjconffile>` Use `<knobjconffile>` as configuration file for universal presenter objects, knobjs (see [6]). `-k-` means use compiled–in default for `<knobjconffile>` (constructed from `<program_name>` by prepending "p" and appending "`.conf`").

`-t<sleeptime>` Use `<sleeptime>` (in seconds) as time interval between two dumps, instead of use compiled–in default 5 seconds.

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

The *cntview* ignores `SIGQUIT` signal. The `SIGHUP` signal causes to reread configuration file (with the same name, as used at startup). The `SIGUSR1` signal stops statistic reading and representing, the `SIGUSR2` continues one. The `SIGINT` signal outputs statistic on printer with compiled–in name by ***lpr(1)*** facility. For user termination in accuracy manner the `SIGTERM` signal must be used.

The *cntview*'s configuration file in the ***knobj.conf(5)*** format (see [6]) can contains only declarations of known objects with types, supported by *cntview*. Such knobj type currently is "`dcnt`" (textual presentation of counters array taked from shared memory).

For the known object "`dcnt`" declaration entry in the ***knobj.conf(5)*** format the first (`name`), third (`type`), fifth (fill dependence), sixth (fill condition), and seventh (clean dependence) fields has its standard in such format meanings.

Second (creator parameters), fourth (filler parameters), eighth (cleaner parameters) and ninth (remover parameters) fields need to complain program interface, provided by "`dcnt`" known functions family.

For example, full declaration for one known object of type "`dcnt`":

```
Obj0041   41;shmid;semid   dcnt \
  41;3;semid;type_ULong;nht,type_String;4;cnt21:cnt22:cnt23 \
  DATA_DAT_0   -   NEVERMORE
```

The ***gen_prescfg (1)*** (see 4.3) generates known object declarations, presented above, from the following prototype entry:

```
dcnt 41 1 -1 shmid semid 3 ULong nht 4 cnt%21N DAT_0 - N
```

4.3 Known object's configuration generator

The known object's configuration generator *gen_prescfg* produces files in the known objects configuration format ***knobj.conf(5)*** (see [6]), used by some SPHERE DAQ utilities (see 4.1, 4.2.1, 4.2.2), from files in the so called known objects prototype format ***knobj.proto(5)***.

```
gen_prescfg [knobjprotofile [knobjconffile]]
```

In the such synopsis form the *gen_prescfg* reads prototype file `knobjprotofile` (or standard input if name omitted), and writes to configuration file `knobjconffile` (or standard output if name omitted).

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

The known objects prototype file consists of zero or more lines, delimited by newline symbols. Lines may be a:

- comment lines,
- preserved 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. Preserved comment lines (lines with "`#`" and "`!`" in the first and second positions) are reproduced to the output untouched.

Other lines must be data lines. Data lines, concatenated with all its continuations, contains one or more fields, separated by space(s) or tab(s) and not contains it. Number of such fields depends on first field contents, which is a known function family name (`fmfunc` here, the same as third field (`type`) in the ***knobj.conf(5)*** format, see [6]). The following `fmfuncs` are supported currently: "`hist`", "`hist2`", "`coord`", "`coord2`", "`cnt`" (for the ***statman(1)*** utility), "`TH1`", "`TH2`" (for the ***histview(1)***), and "`dcnt`" (for the ***cntview(1)***).

For `bname` and `btitle` fields `%N`, `%p` and `%P` conversions in the ***printf(3)*** style are supported, for `fillcnd` field – `%p` only.

`%<num1>.<num2>N` replaced by number, starting from `<num1>` and incremented once per `<num2>` names generated.

`%p` replaced by polarization mode symbolic representation, namely "`p`" for plus polarization mode, "`m`" for minus polarization mode, "`z`" for mode without polarization (so called zero), "`b`" for bad (unknown for some reasons) polarization mode – cyclically in such order.

%<num1>.<num2>P replaced by number, starting from <num1> and followed by polarization mode symbolic representation as for %p. Number incremented once per four names generated (for example: cnt0p, cnt0m, cnt0z, cnt0b, cnt1p and so one). <num2> is ignored.

If bname value contains the "#" as first symbol, all generated known object entries will be commented out in the output file.

Other fields are specific for fmfuncs mentioned above.

4.4 Control over SPHERE DAQ

For control over SPHERE DAQ a supervisor control module (see [2]) with Graphics User Interface (GUI), which was designed to be self–explanatory, can be used. Supervisor uses ***sv.conf(5)*** configuration file, which is a makefile in the current *qdpb*'s implementation.

Because of a some "manual control" is useful from time to time, users can ***make(1)***'ing *sv.conf*'s targets directly without *sv*'s mediation. Target and variable names are selected to be self–explanatory (for details see an actual *sv.conf* file).

At least following targets are defined in the *sv.conf*:

load – load and configure CAMAC and branch point kernel modules, load *bpget* service module and connect it (into BPRUN state) to branch point directly.

unload – unload *bpget* service module, unload CAMAC and branch point kernel modules (counterpart for load target).

loadw – load *writer* with requests of mandatory parameters and mentions about optional ones, and connect it (into BPSTOP state) to branch point directly.

unloadw – unload *writer* (counterpart for loadw target).

loads – load *statman* and connect it (into BPSTOP state) to branch point directly.

unloads – unload *statman* (counterpart for loads target).

loadc – load *cntview* in own *xterm* window.

unloadc – unload *cntview* (counterpart for loadc target).

loadh – load *histview* in own *xterm* window.

unloadh – unload *histview* (counterpart for loadh target).

start_all – change state of all branch point connections to BPRUN.

stop_all – change state of all branch point connections to BPSTOP (counterpart for start_all target).

init – initialize read–out CAMAC modules. (Also is a part of the load . As standalone used after read–out crates power cycle.)

finish – deinitialize read–out CAMAC modules (counterpart for init target).

continue – start CAMAC interrupts handling.

pause – stop CAMAC interrupts handling (counterpart for continue target).

cleanall – clean whole *statman*'s statistic, collected in a shared memory.

clean – clean *statman*'s statistic in accordance with configuration file */scratch/qqq3/etc/istatman*<RUN>*.conf*.

pauseh – pause *histview* data visualization (counterpart for conth target).

pausec – pause *cntview* data visualization (counterpart for contc target).

`conth` – continue *histview* data visualization.
`contc` – continue *cntview* data visualization.
`status` – view status of DAQ system parts.
`seelog` – start viewing of DAQ system messages, logged by ***syslogd(8)***.
`confs` – reinitialize *statman* after change of it's configuration file(s).

5 SPHERE DAQ system in work

5.1 Testing on the single–crate stand

Here we describe some SPHERE DAQ system tests, performed at autumn 1999. Hardware configuration was as follows:

- computer with CPU AMD K6 200 MHz, RAM 64 Mb;
- PK009 interface card + KK009 CAMAC crate controller [5];
- one CAMAC crate with one sample of each CAMAC hardware modules, planned to use in real experiments.

First of all some speed tests using two versions of *speedtest* CAMAC kernel module – implemented on kernel context subroutines of the CAMAC facility ***camac(4)*** (see also [3]) and on kernel interface to KK009/KK012 memory (see [3]) – were performed. The function `hand()` – interrupt handler itself – was contain: in the first case – 1 "select crate" command (internal for KK0xx crate controller), 10 CAMAC read commands, and 1 drop LAM command; in the second case – 10 CAMAC read commands and 1 drop LAM command, each with "select crate" and some other costs of CAMAC facility working. CAMAC interrupts was generated with 20 kHz frequency. CAMAC driver ***kk(4)*** was in the fast interrupt mode. Interrupt reaction time (including "select crate" command performing) and time for a one single–cycle CAMAC command performing were measured by oscilloscope directly on ISA bus (using ISA slot contacts). Results are summarized in the following Table:

computer base frequency × CPU multiplier	75 × 3	66 × 3
PCI frequency	75/2	66/2
ISA frequency	9.4 MHz	8.35 MHz
camac(9) *speedtest* implementation		
Interrupt reaction	3.0–3.8 mks	3.4–4.5 mks
Single–cycle CAMAC command	4.4 mks	5.0 mks
kk(9) *speedtest* implementation		
Interrupt reaction	2.8–3.8 mks	3.0–4.1 mks
Single–cycle CAMAC command	2.0 mks	2.3 mks

Note, that up to 30 kHz system not seen very loaded – interactive working was comfortable enough.

For whole SPHERE DAQ system testing the `zzz99` RUN was configured, trigger emulation was designed, and *hand* CAMAC interrupt handler (see 2.2) was written. Working software configuration consists of:

- running supervisor, by which was:
- loaded branch point and *hand* kernel modules,
- one branch point output stream consumed by ***writer(1)***, and

- one branch point output stream consumed by ***statman (1)***, which maintains 66 "native" and 3 cell calculated histograms and 24 counters (normal bandwidth for SPHERE experimental data) in a shared memory;
- 5 "native" and 3 cell calculated histograms was visualized by ***histview (1)*** with 5 second interval, and
- 24 counters was printed on screen by ***cntview (1)*** with the same interval.

Triggers was initiated by generator with 2 kHz frequency ($\approx$ 1000 triggers per "accelerator burst" is our expected optimum for real experiments). System works non–stop more than 5 hours (mainly without writing to disk due to limited space) successfully, and interactions between base elements was correct.

5.2 Using on the SPHERE experiment

The SPHERE DAQ system was tested under real work conditions during four accelerator RUNs at 2002 and one at 2003. First of it was March'2002 RUN of Dubna Nuclotron. Hardware configuration of SPHERE setup for this RUN was as follows:

- computer with CPU AMD K6–III 400 MHz, RAM 128 Mb;
- single CAMAC branch organization: PK009 interface card + three KK009 CAMAC crate controllers [5];
- read–out electronics (mainly) consists of 28 ADC channels, 28 TDC channels, 10 hodoscopic register channels (all are 16–bit values) and 28 scaler channels (32–bit values); corresponding CAMAC modules are situated in three crates.

Three event types was collected and corresponding three packet types was produced: `DATA_CYC_BEG` – accelerator burst begin, `DATA_DAT_0` – trigger of type 0, and `DATA_CYC_BEG` – accelerator burst end, with corresponding packet body lengthes of 6, 174, and 86 bytes (packet header length is fixed and equals to 40 bytes, see [2] for packet format details). Because of initially such binary format was prepared for June'2001 RUN of Synchrophasotron, it named "`jun01`". Working software configuration consists of:

- loaded branch point and *hand* kernel modules,
- one branch point output stream consumed by ***bpget (1)*** (permanent client),
- one branch point output stream consumed by ***writer (1)*** (during runs with datafiles writing), and
- one branch point output stream consumed by ***statman (1)*** (during approximately whole time), which maintains 66 "native" and 3 cell calculated histograms, 28 "native" and 4 ***statman (1)***'s internal counters in a shared memory;
- 25 "native" histograms was visualized by ***histview (1)*** (ROOT [4] implementation) with 5 second interval, and
- 28 "native" and 3 ***statman (1)***'s internal counters was printed on screen by ***cntview (1)*** with the same interval.

The SPHERE DAQ system works stably and reliably during approximately 60 hours (40 hours continuously). Approximately 240 Mbytes of datafiles was written.

So a DAQ changes history can be summarized as following:

Binary format	Hardware	Software			Working time, hour	Data acquired, Mbytes
		statman	*histview*	*cntview*		
March'2002						
jun01 DATA_CYC_BEG 6 bytes, DATA_DAT_0 174 bytes, DATA_CYC_END 86 bytes	28 ACD ch., 28 TCD ch., 10 hod. reg. ch., 24 scal. ch. (3 readout crates)	66 "native" hist., 3 cell calc. hist., 28 "native" cnt., 4 internal cnt.	25 "native" hist.	28 "native" cnt., 3 internal cnt.	≈ 60	≈ 240
June'2002						
jun01	The same	66 "native" hist., 6 cell calc. hist., 28 "native" cnt., 4 internal cnt.	71 "native" hist.	28 "native" cnt., 3 internal cnt.	≈ 94	≈ 800
November'2002						
jun01	online computer upgrade: CPU AMD K7–550 MHz and RAM 256 Mb	66 "native" hist., 6 cell calc. hist., 16 cond. fill. hist., 28 "native" cnt., 153 cell calc. cnt., some cell calc. fill. cond.	71 "native" hist.	28 "native" cnt., 153 cell calc. cnt.	≈ 170	≈ 536
December'2002						
dec02 DATA_CYC_BEG 6 bytes, DATA_DAT_0 308 bytes, DATA_CYC_END 86 bytes	68 ACD ch., 68 TCD ch., 5 hod. reg. ch., 20 scal. ch. (3 readout crates)	141 "native" hist., 2 cell calc. hist., 24 "native" cnt., 80 cell calc. cnt., 80 cell calc. fill. cond.	≈ 50 "native" hist.	24 "native" cnt., 80 cell calc. cnt.	≈ 30	≈ 600

June'2003						
dec02	The same	150 "native" (1D and 2D) hist., 4 cell calc. hist., 24 "native" cnt., 100 cell calc. cnt., 170 cell calc. fill. cond.	varied as needed	varied as needed	≈ 80	≈ 6800

6 Software dependencies and portability

All notes about software dependencies and portability, issued for the *qdpb* system [2], are applicable for its subset, the SPHERE DAQ system. Ones for SPHERE *offline* system presented in details in [6].

Acknowledgements

Author have a pleasure to thank to S.G.Reznikov for numerous fruitful discussions, essential help in testing on stand, and some code contribution; and to A.G.Litvinenko for March'2002 RUN trigger design and essential help in testing on SPHERE setup during March'2002 RUN.

Investigations was supported in partial by the RFBR grant N 02–02–16024.

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] Gritsaj K.I. Private communications.

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

[5] Churin I., Georgiev A. Microprocessing and Microprogramming, 23 (1988), p.153.

[6] Isupov A.Yu. **Configurable data and CAMAC hardware representations for implementation of the SPHERE DAQ and offline systems.** Communication of JINR E10–2001–199, Dubna, 2001.

Received on October 6, 2003.

Исупов А. Ю.
E10-2003-187

Реализация сбора и обработки данных установки СФЕРА
на основе системы *qdpb*

Описывается архитектура системы сбора данных установки СФЕРА (ЛВЭ ОИЯИ), построенной на основе системы обработки данных с точками ветвления (*qdpb*) и конфигурируемых представлений экспериментальных данных и аппаратуры КАМАК. Рассматривается реализация программного кода систем сбора и обработки данных, зависимого от конфигурации аппаратуры и характера экспериментальных данных установки СФЕРА. Также описаны специфические для данной реализации программные модули.

Работа выполнена в Лаборатории высоких энергий им. В. И. Векслера и А. М. Балдина ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна, 2003

Isupov A. Yu.
E10-2003-187

SPHERE DAQ and Off-Line Systems:
Implementation Based on the *qdpb* System

Design of the on-line data acquisition (DAQ) system for the SPHERE setup (LHE, JINR) is described. SPHERE DAQ is based on the *qdpb* (Data Processing with Branchpoints) system and configurable experimental data and CAMAC hardware representations. Implementation of the DAQ and off-line program code, depending on the SPHERE setup's hardware layout and experimental data contents, is explained as well as software modules specific for such implementation.

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

Communication of the Joint Institute for Nuclear Research. Dubna, 2003

Макет *Т. Е. Попеко*

Подписано в печать 24.10.2003.
Формат 60 × 90/16. Бумага офсетная. Печать офсетная.
Усл. печ. л. 1,25. Уч.-изд. л. 1,78. Тираж 290 экз. Заказ № 54149.

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6.
E-mail: publish@pds.jinr.ru
www.jinr.ru/publish/