E10-2001-116

K.I.Gritsai, A.Yu.Isupov

A TRIAL OF DISTRIBUTED PORTABLE DATA ACQUISITION AND PROCESSING SYSTEM IMPLEMENTATION: THE *qdpb* — DATA PROCESSING WITH BRANCHPOINTS

1 Motivation and introduction

Data acquisition (DAQ) and slow control systems based on electronics in the CAMAC standard and computers with IBM PC compatible architecture are widely used for physics experiments with not so high speed of data acquisition and transfer at present time. Such systems are widely distributed due to a great assortment of a CAMAC hardware modules, PC adapters for CAMAC crate controllers availability, and relatively low cost of a such equipment.

Operating system (OS) used on the online computer dictate a DAQ system design and organization, so adequate OS selection can essentially simplify (inadequate – strongly complicate) implementation, maintenance, and using of a DAQ system.

An UNIX–like OS's are optimal for a wide range of DAQ and slow control jobs. UNIX is a multiprocess and multiuser OS with powerful mechanisms for interprocess and inter–computer communications, support of advanced networking and graphics interfaces, extended tools for software design. Costs for UNIX working itself are very small and negligible for PC beginning from i486 or Pentium class CPU's. High portability of UNIX programming, availability of free distributable UNIX–like OS's, and approximately unlimited quantity of an existed software are also very attractive. One of such free UNIX–like OS's, enough stable, reliable, modern, and dynamically developed simultaneously due to it's very weighted up design policy, is FreeBSD.

On the other hand, a lack of distributed portable data acquisition and processing system, enough flexible for working with a wide range of nuclear electronics hardware, and enough computer platform independent, is observed. Described *qdpb* system is a trial to fill this gap.

Through the following text the references to terms are highlighted as **boldface text**, file and software package names – 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 3.7.1. Subjects of substitution by actual values are enclosed in the angle brackets: `<pidfile>`. All mentioned trademarks are properties of its respective owners.

2 Formal description

This section gives an overview of the distributed portable data acquisition and processing system, *qdpb* (for "*d*ata *p*rocessing with *b*ranchpoints").

2.1 Terminology

Here we present a short dictionary of the *qdpb* system specific terms, and collects references to its descriptions. Generic UNIX and hardware terms does not covered here (see, for example, [1], [2]).

Branch point – source and/or termination end(s) for some **packet streams**. Branch point implementation depends on neither **packet stream** contents, nor **packet**

structure. See 2.2 and 3.6.

CAMAC driver is a kernel device driver, pseudo–device or loadable kernel module, and deals with some specific CAMAC hardware (such as CAMAC crate controller). See 2.2, [5].

CAMAC interface (or **facility**) **in kernel** is an OS kernel's software object, implemented as pseudo–device, which provides support for software objects, mentioned here as **CAMAC drivers** and **CAMAC modules**. See [5].

CAMAC kernel module is a loadable kernel module (therefore can be user supplied easily), and implements handlers for CAMAC interrupts, produced by some specific CAMAC hardware (such as CAMAC crate controllers). See 2.2, 3.2, [5].

CAMAC process(es) – process(es) works with CAMAC in user context. Its destined for some slow operations (CAMAC primary initialization, slow control, debugging, tests, etc.). See 2.2, 3.3, [5].

CAMAC subsystem contains:

- **CAMAC interface in kernel**,
- **CAMAC driver**(s),
- **CAMAC kernel module**(s),
- **CAMAC process**(es).

Control module – process, which destined for control over some *qdpb* system element(s). It does not deals with **packet streams** at all. It usually launched by **supervisor** (see 3.7.1). See 2.2 and 3.7.

Cycle of accelerator – one beam burst + following pause (for example, 0.5 sec + 9 sec for Dubna Synchrophasotron).

Data acquisition and processing system (*qdpb* itself) – union of the **hardware subsystem** and the **data processing subsystem**. See 2.2.

Data file – events information in the form of **packet** sequence, written continuously on anything external storage media (hard disk, magnetic tape, etc.). Writing performed by the data files producer **work module** (see 3.4.1) and satisfy to the following conditions:

1. **events** from the same accelerator cycle doesn't appear in the different files;
2. **event** sequence in the data file is the same, as in the **packet stream** from nearest downstream **branch point** (relative to data files producer);
3. single file doesn't contain **events** from a different **runs**.

See 2.2.

Data processing subsystem contains elements of four types:

1. **work module**(s) (see 3.4),
2. **service module**(s) (see 3.5),
3. **control module**(s) (see 3.7),
4. **branch point**(s) (see 3.6) (and/or **event merger**(s) (see 3.8)),

connected by **packet streams**. See also 2.2.

Event – information, logically grouped together for some reasons, for example, obtained as result of one CAMAC interrupt handling, etc. Events in the **data processing subsystem** represented by **packets**, where event information stored in the packet body. See 2.2.

Event merger [1] – some flavour of **branch point**, which modify packet structure, so each one output packet produced by concatenation of packet body(ies) of corresponding input packet(s). Term "packets correspondence" is implementation dependent. See 2.2 and 3.8.

Hardware subsystem – union of all subsystems, deals with hardware directly. **CAMAC subsystem**, in particular, is a part of it.

Interface to operator is a **control module**, identical to **supervisor** or launched by it. It performs a dialogue between operator and supervisor.

Offline system[2] is a part of the **data processing subsystem** and destined to group together all code, dependent on experimental data contents, which tends to be changed between different accelerator **RUN**s. .

Packet – some number of bytes. It contains the packet header and the packet body. See 2.2 and 3.1.

Packet classification based on value of **packet type**, field in the packet header. **Packet**s can be of data type (in the current implementation – only such), control type, etc. See 3.1.

Packet consumer – **work module**, which terminates the **packet stream**. It reads from standard input and writes information by some system call. See 2.2, 3.4.

Packet filter – **work module**. It reads **packet stream** from the standard input, modifies its contents and writes it to the standard output. See 2.2 and 3.4.

Packet source – kernel module or **work module**, which produces **packet**s by using `make_pack()` and `crc_pack()` library functions or kernel calls (see 3.1), and puts it in the **packet stream** (usually standard output) or in the **branch point**'s buffer directly. Packet source can reside on the local or remote (exclude kernel module packet source) host relative to a **branch point**. See 2.2, 3.2, 3.4.

Packet (data) stream is a packet sequence. It can be:

- input – packet stream(s) to **branch point**. It(s) are different.
- output – packet stream(s) from branch point. It(s) are identical.

See 2.2 and 3.6.

Packet type represented by packet header variable `u_short type`. See 2.2, 3.1.

(Experimental) Run (in the narrow meaning) – some number of **accelerator cycles** (subset of the total cycles sequence), performed under identical experimental conditions. Usually represented by some **data files** sequence.

[1] Authors introduce term "event merger", because it more close to proposed (too simple) functionality, than known term "event builder".

[2] Offline system depends on experimental setup strongly, so it's description doesn't fit in this generic paper.

RUN (in the wide meaning) – accelerator run. In the **offline system**[2] RUN name (denoted as `<run_name>` below) produced by concatenation of (shortened) month name and year number. .

Service module – process, which destined for **packet streams** management and does not modifies the **packet stream** contents. Service module implementation depends on neither **packet stream** contents, nor **packet** structure. See 2.2 and 3.5.

Supervisor – **work** or **control module**, which performs control over *qdpb* system elements by mechanisms, available in such elements. See 2.2 and 3.7.1.

Work module – utility process, which can modify **packet stream** contents, but not **packet** structure. Work module can be:

- a **packet source**,
- a **packet consumer**,
- a **packet filter**.

It can be either setup/RUN independent (see 2.2 and 3.4) or dependent.

2.2 Conception

Here we declare basic principles of the *qdpb* system design and working.

(**Event**) information in the **data processing subsystem** represented by (data) **packet**s. Each packet contains the packet header and the packet body. Packet header have fixed size and format and contains at least the following fields: packet identifier, packet length, packet type, packet serial number, packet creation time and packet check sum (CRC). Packet identifier is identical for all packets. Packets of the different types have separate serial numeration. Repetition of packets with the same number is permitted, but all such packets except first obtained are ignored. Packet type do not coupled with packet length. Packet size limited by the `PACK_MAX` value. See also 3.1.

(**Event**) information in the **data processing subsystem** distributed as **packet stream**s. Packet stream is a packet sequence. It can contain packets of different types, order of packets in stream is arbitral. Stream is implemented as non–structured byte sequence without positioning. Packet origin search in the stream based on the calculation of the packet offset from the stream origin, and verified by packet identifier. Reaction on the packet origin search error is implementation dependent. Recommended reaction as follows: search for packet origin by packet identifier and discard all stream contents from place, where error was recognized, up to the founded packet origin. The library functions (see 3.1) is provided for dealing with packets (exclude its bodies) and packet streams.

Data processing subsystem consists of **work modules**, **control modules**, **service modules**, and **branch point**(s).

Hardware subsystem consists of subsystems, deals with hardware directly. **CAMAC subsystem**, in particular, is a part of it.

CAMAC crate controller(s) with interface card(s) for some computer architecture (for example, for ISA or PCI bus) and corresponding software can be considered as a CAMAC hardware subsystem. As software implementation we originally take the *camac* package, described in [3] and [4], which supports only adapters PK009/PK012 for CAMAC crate controllers KK009/KK012 (see [6]/[7]), designed by I.N.Churin in the JINR, LNP. Currently we use its successor, the *camac2* package [5], which supports also Churin's KK011 [8], KKL01 [9] from N.I.Lebedev (JINR, LPP), and CCPC4 [10], CCPC5 from S.N.Basilev (JINR, LHE), and can be easy extended to support of anything other CAMAC adapter/controller pairs.

CAMAC subsystem consists[3] of **CAMAC interface in kernel**, **CAMAC driver**(s), **CAMAC kernel module**(s), and **CAMAC process**(es).

The *qdpb* system expected to be very flexible in accommodation of a new hardware. Generally speaking, only availability of adapters for your selected computer architecture and specific software for dealing with it (or ability to design it by yourself) are required for attaching to the *qdpb* system framework.

Data acquisition and processing system (*qdpb* itself) is a union of the **hardware subsystem** and the **data processing subsystem**.

Work module is a process, destined for information processing. It can read from the packet stream and/or write to the packet stream. Input and output stream contents may in general case be nonidentical. Work module can contains mechanisms for control over it (signal handling, control packets interpretation, etc.). Work module can be a **packet source**, **packet consumer**, or **packet filter**. Recommended the following agreement for the work module implementation: reading and writing are performed as buffered input/output from/to standard input/output with blocking; `SIGPIPE` signal and `EOF` state are follows to the process termination. See also 3.4.

Control module is a process, destined for control over some *qdpb* system element(s). It does not deals with packet streams at all. It usually launched by supervisor. It can be either setup/RUN independent (see 3.7) or dependent.

Service module is a process, destined for stream management. It can read from the packet stream or write to the packet stream. Input and output packet sequences are identical. Service module implementation depends on neither packet stream contents, nor packet structure, so it is universal. See also 3.5.

Branch point is a source and/or termination end(s) for some packet streams and destined for producing some identical output packet stream(s) from some different input one(s). Branch point does not modify packet contents, so its implementation depends on neither packet stream contents, nor packet structure, it is universal. Order of packets from different inputs is arbitral in output, but packets order of each input still the same in the output stream. Branch point also implements packet buffer and control facility for itself. Recommended the branch point implementation as part of the OS kernel (kernel module or driver), provides specific system call(s). See also 3.6.

Event merger is a some flavour of the **branch point**. It destined for producing some identical output packet stream(s) from some different input one(s). Event merger modify packet structure as follows: each one output packet produced by making new packet header, and concatenating of packet body(ies) of one or more so called "corre-

[3]*camac2* package description doesn't fit in this generic paper.

sponding" input packets (one from each registered input stream – input channel). In the current implementation (see also 3.8) the correspondence between input and output packets requires:

- input and output packet types (`header.type`) correspondence, declared for each input channel at its registration, and
- input packet numbers (`header.num`) coincidence for candidates from all input channels.

Packets with types, which have not declared correspondence, does not taken from the input channels. Packets with numbers, which have not corresponders in all input channels, are discarded. Event merger implementation not depends on packet stream contents. Event merger also implements control facility for itself. Recommended the event merger implementation as part of the OS kernel (kernel module or driver), provides specific system call(s). See also 3.8.

Supervisor is a work or control module. It performs at least start, stop and control actions over *qdpb* system by operator commands. Command to action correspondence described in the supervisor configuration file ***sv.conf(5)***. *qdpb* system elements, which can be controlled, are: kernel elements (kernel module(s) of hardware subsystem, branch point(s), event merger(s)) – by means of its specific system call(s); work module(s) – by means of its specific mechanisms (see above). Not provided: control over other *qdpb* system elements; reaction on states of the *qdpb* system. For remote control supervisor launches control modules on the remote host by means of ***rsh(1)***, ***ssh(1)*** or ***rcmd(3)***. See also 3.7.1. Some *qdpb* system elements (if its have own Graphics User Interface (GUI)) can be controlled by operator directly instead of through the supervisor (for example, data presentation control modules).

3 Implementation notes

Here we describe parts of the *qdpb* system with more implementation details.

3.1 Packets

In the current implementation packet is:

```
typedef struct {
        header head;
#define DATA_MAX        (sizeof(header) * 99)
        char data[DATA_MAX];
} packet;
```

where packet body `data[DATA_MAX]` have a format, not significant for data processing subsystem (exclude some setup/RUN dependent work modules). `header` have the following format:

```
typedef struct {
#define ID_LEN          16
```

```
  char id[ID_LEN]; /* identifier, 16 bytes, = "Packet begin >>>" */
  int len;           /* packet length, 4 bytes */
  u_long crc;     /* checksum over rest header and data, 4 bytes */
  struct timeval tv; /* packet "originating" time, 4+4 bytes */
  u_short flag;      /* header flags, 2 bytes */
  u_short type;      /* packet type, 2 bytes */
  u_long num;        /* packet number, 4 bytes */
} header;
```

and `sizeof(header)` is equals to 40 bytes.

In the current implementation interfaces for user (as part of the *qdpb* library) and kernel contexts are provided.

`crc`, `make_pack`, `crc_pack`, `merge_pack`, `write_pack`, `rawrite_pack`, `read_pack` – packet handling routines (user context interface).

```
#include <packet.h>
u_long crc(void *ptr, int len)
u_long CRC_CHECK(packet *pack, int flags)
int make_pack(const header *head, const char *data, int len,
        packet *pack, int flags)
void crc_pack(packet *pack)
int merge_pack(const header *head, const char **data, int *len,
        int num, packet *pack, int flags)
int write_pack(FILE *stream, const packet *pack)
int rawrite_pack(int fd, const packet *pack)
int read_pack(FILE *stream, packet *pack, int flags)
```

The `crc()` function computes a POSIX 1003.2 checksum for data with length `len`, pointed by `ptr`, and returns it.

Macro `CRC_CHECK()` computes checksum for packet, pointed by `pack`, if requested by `flags` (possible values are defined in the *packet.h* header file), and returns it. Otherwise current contents of `header.crc` field will be returned.

The `make_pack()` function makes packet from header, pointed by `head`, and data with length `len`, pointed by `data`, in the destination, pointed by `pack`. `make_pack()` fills only `header.id` and `header.len` (and, if requested by `flags`, `header.tv`, `header.crc`, and `header.flag`) fields. Fields `header.type` and `header.num` must be already filled by user, as well as `data`. If data length `len` > `DATA_MAX`, defined in the *packet.h* header file, data truncated to fit `DATA_MAX`. `make_pack()` returns length of resulted packet.

The `crc_pack()` function computes checksum for packet, pointed by `pack`, fills `header.crc` field and modifies `header.flag` field accordingly.

The `merge_pack()` function makes packet from header, pointed by `head`, and `num` pieces of data, pointed by array `data` of pointers to characters, with lengths, stored in array `len` of integers in the destination, pointed by `pack`. `merge_pack()` fills only `header.id` and `header.len` (and, if requested by `flags`, `header.tv`, `header.crc`, and header.flag) fields. Fields header.type and header.num must be already filled by user, as well as all data pieces. merge_pack() returns length of resulted packet.

The write_pack() function writes packet, pointed by pack, into output stream stream.

The rawrite_pack() function writes packet, pointed by pack, into already opened file descriptor fd.

The read_pack() function reads packet into destination, pointed by pack, from input stream, pointed by stream, always with checking of header.id and, if required by flags, header.crc fields correctness. read_pack() searches for valid (determined by valid header.id and, if required by flags, by correct header.crc) packet, if error on input stream take place. (All data, obtained before that search will be successful, are discarded).

The crc() and CRC_CHECK() returns positive value.

The make_pack() returns nonnegative value.

The merge_pack() returns nonnegative value if success, and −1 with global variable errno setting otherwise.

The write_pack() and rawrite_pack() returns 1 if successful and negative value with global variable errno setting otherwise.

The read_pack() returns 1 if packet read successful, 0 if EOF reached, and negative value with global variable errno setting otherwise.

[EIO] Write packet failed in function write_pack() due to any errors specified for the routine *fwrite(3)* or *fflush(3)*.

[EIO] *ferror(3)* detects stream error in function read_pack().

[EPIPE] Incorrect number of bytes read in function read_pack().

[EMSGSIZE] Too big length of resulted packet in function merge_pack().

The rawrite_pack() may fail and set errno for any of the errors specified for the routine *write(2)*.

crc, make_pack, crc_pack, merge_pack – kernel interface to packet handling.

```
#include <packet.h>
u_long crc(void *ptr, int len)
u_long CRC_CHECK(packet *pack, int flags)
int make_pack(const header *head, const char *data, int len,
        packet *pack, int flags)
void crc_pack(packet *pack)
int merge_pack(const header *head, const char **data, int *len,
        int num, packet *pack, int flags)
```

These functions can be compiled into kernel as interface to packet handling for other parts of OS kernel (for example, event merger, described in 3.8, or user CAMAC interrupt handler).

The crc() function computes a POSIX 1003.2 checksum for data with length len, pointed by ptr, and returns it.

Macro `CRC_CHECK()` computes checksum for packet, pointed by `pack`, if requested by `flags` (possible values are defined in the *packet.h* header file), and returns it. Otherwise current contents of `header.crc` field will be returned.

The `make_pack()` function makes packet from header, pointed by `head`, and data with length `len`, pointed by `data`, in the destination, pointed by `pack`. `make_pack()` fills only `header.id` and `header.len` (and, if requested by `flags`, `header.tv`, `header.crc`, and `header.flag`) fields. Fields `header.type` and `header.num` must be already filled by user, as well as `data`. If data length `len` > `DATA_MAX`, defined in the *packet.h* header file, data truncated to fit `DATA_MAX`. `make_pack()` returns length of resulted packet.

The `crc_pack()` function computes checksum for packet, pointed by `pack`, fills `header.crc` field and modifies `header.flag` field accordingly.

The `merge_pack()` function makes packet from header, pointed by `head`, and `num` pieces of data, pointed by array `data` of pointers to characters, with lengths, stored in array `len` of integers, in the destination, pointed by `pack`. `merge_pack()` fills only `header.id` and `header.len` (and, if requested by `flags`, `header.tv`, `header.crc`, and `header.flag`) fields. Fields `header.type` and `header.num` must be already filled by user, as well as all data pieces. `merge_pack()` returns length of resulted packet.

The `crc()` and `CRC_CHECK()` returns positive value.

The `make_pack()` returns nonnegative value.

The `merge_pack()` returns nonnegative value on success, and `-EMSGSIZE` for too big length of resulted packet.

3.2 Kernel modules of hardware subsystem

Loadable kernel module is more flexible method for interrupt handler implementation under Unix–like OS's (compare, for example, with compiled–in code for monolithic kernel). Kernel module of hardware (in particular CAMAC) subsystem implements a user handler of the hardware (CAMAC) interrupt. It also implements specific system call(s), in particular control facility for itself. The control modules ***handoper(8)*** and ***handconf(8)***, based on such call(s), used as, respectively, control and configuration/testing utilities for the CAMAC kernel module. Details about requirements to CAMAC kernel module and toolkit for its implementation can be found in the *camac2* package [5]. See also 3.6, 3.1.

3.3 CAMAC processes

CAMAC process(es) destined for "asynchronous" CAMAC operations – CAMAC primary initialization, slow control, debugging, tests, etc. – which neither requires high performance nor coupled with interrupt handling, and works in the user context. In particular, in the *camac2* package [5] there are ***crate(1)***, ***c_master(1)***, ***naf(1)*** utilities.

Its uses library routines, based on the user context subroutines, implemented by the CAMAC interface in kernel.

3.4 Work modules

Provided the following packet sources:

- packet stream generator ***genpack(1)*** (for debugging purposes);

the following packet consumers:

- packet stream dumpers ***readpack(1)***, ***readpack2(1)*** (for debugging purposes),
- data files producer (see 3.4.1);

the following packet filters:

- software filter ***filter(1)*** (see also ***filter.conf(5)***).

3.4.1 Data files producer

Data files producer (in the current implementation named *writer*) is a work module of the packet consumer type, which writes packet stream contents to file(s) on anything external storage media.

```
writer [-l] [-t] [-b<bpemstat> [-e]] [-r{-|<runname>}]
       [-d{-|<outdir>}] [-s{-|<#>}] [-j{-|<jobfile>} [-J]] [-f<#>]
       [-p{-|<pidfile>}] [-u"<Comment string>"] [-m]
```

In the such synopsis form the *writer* reads packets from standard input and writes all obtained data packets in a datafile of (approximately) 400 kbytes size, and name, constructed from string "`test`", in the directory */data/tmp*. After datafile closing next datafile opened, and so on.

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

`-b<bpemstat>` Use branch point (see 3.6) as input instead of standard input, and open it at `<bpemstat>`[4] state. (Available only for systems, where branch point is implemented).

`-e` Open event merger (see 3.8) at `<bpemstat>` state and read packets from it instead of branch point. (Available only for systems, where event merger is implemented).

`-r<runname>` Use `<runname>` as generic part of datafile(s) name, instead of default. `-r-` means use compiled–in default for `<runname>`.

`-d<outdir>` Use `<outdir>` as name for directory, where current (and all complete, if job not move it) datafile(s) will be stored, instead of default. `-d-` means use compiled–in default for `<outdir>`.

`-s<#>` `<#>` is a recommended for *writer* size of datafile(s) (in bytes), instead of compiled–in default.

[4]`<bpemstat>` need to be substituted by `r` for "run", `s` for "stop", or `d` for "discard".

`-j<jobfile>` Read job description (see below for more information) from file named `<jobfile>` instead of default. `<jobfile>` must be specified with full absolute path because of lack of search path machinery in the current implementation of *writer*. `-j-` means use compiled–in default for `<jobfile>`. Job will be performed after each datafile completion.

`-J` Perform job by `exec()`'ing instead of `system()`'ing by default. (In other words, *writer* wait for job completion by default and doesn't wait, if `-J` supplied).

`-f<#>` `<#>` is a `pack_flags` value for the `read_pack()` function (see 3.1).

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

`-u"<Comment string>"` (Double or single quotes needs only to avoid a shell's substitutions). Write packet with `<Comment string>` as body of first packet in each produced datafile.

`-m` Write packet with identification of machine, on which *writer* run, as body of second packet in each produced datafile.

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

The *writer* ignores `SIGHUP`, `SIGINT` and `SIGQUIT` signals. For user termination in accuracy manner `SIGTERM` signal must be used.

A job description file for *writer* is a shell script with some commands, which need to be executed after completion of each datafile. This job file must return right exit code for interpretation by *writer*. Inside job the name of last completed datafile available as first positional parameter ($1). For example:

```
#!/bin/sh
# Newest complete datafile available here as $1
mv $1 /tmp
exit $?
```

3.5 Service modules

Required at least four types of service modules:

1. server end of a socket connection;
2. client end of a socket connection;
3. ***bpput(1)*** puts packets from stream into branch point buffer;
4. ***bpget(1)*** gets packets from branch point buffer to stream;

In the current implementation ***faucet (1)*** and ***hose (1)*** utilities from ***netpipes (1)*** package are used as service modules of types 1 and 2, respectively.

3.6 Branch point implementation

In the current implementation branch point is a loadable kernel module of syscall type and must be loaded at syscall offset `BPSYS` (defined in the *branchpoint.h* header file). It provides access to packet buffer for so called work and service modules – utilities for packet streams processing in the *qdpb* system. It implements user and kernel context interfaces described below.

`bpopen, bpclose, bpget, bpput, bpgetstat, bpsetstat, bpclear` – user context interface to branch point kernel module.

```
#include <bpio.h>
int bpopen(int flag)
int bpclose(void)
int bpget(void *buf)
int bpput(const void *buf, int len)
int bpgetstat(struct bpargs *bpargs)
int bpsetstat(const struct bpargs *bpargs)
int bpclear(void)
```

The `bpopen()` function opens branch point for caller process with options, defined by `flag` argument. Options can constructed by OR'ing ("|") one of the following possible operations, defined in the *bpio.h* header file:

BP_GET – open for get packets,

BP_PUT – open for put packets;

and one of the following possible states:

BP_RUN – open at run state,

BP_STOP – open at stop state,

BP_DISCARD – open at discard state.

The `bpclose()` function closes branch point for caller process.

The `bpget()` function gets one packet from branch point's buffer into destination, pointed by `buf`, and returns length of obtained packet.

The `bpput()` function puts one packet with length `len` from source, pointed by `buf`, into branch point's buffer.

The `bpgetstat()` function gets branch point status into `bpargs` structure, defined in the *branchpoint.h* header file as follows:

```
struct bpargs {
        pid_t p_pid;
        int oper;
        int stat;
};
```

Before call `bpargs` must contains valid (positive) `p_pid`. After call all fields will be filled and valid.

The `bpsetstat()` function sets branch point status from filled by user source, pointed by `bpargs`. Before call `bpargs` must contains valid `stat` (BP_RUN, BP_STOP, or BP_DISCARD) and `p_pid` (nonnegative) fields. Zero `p_pid` means sets status for each registered process with user supplied `oper` (valid are BP_GET, BP_PUT, or its OR'ed combination means all registered processes).

The `bpclear()` function clears branch point's buffer.

The `bpget()` returns 0 at EOF condition, all other functions returns 0 if successful. Otherwise −1 returned and global variable `errno` is set to indicate the error.

[EAGAIN] Free space in branch point buffer not available (`bpput()` function).

[EACCES] Branchpoint not opened (`bpget()`, `bpput()`, and `bpclose()` functions).

[EMSGSIZE] Incorrect packet size specified in `bpput()` function.

[EPIPE] BP_PUT operation failed due to branch point not opened for BP_GET by anybody (bpput() function).

[EINVAL] Invalid state requested in bpopen() function; invalid bpargs specified in bpgetstat() or bpsetstat() functions.

[ENOBUFS] Space in table of registered process exceeded (bpopen() function).

[ESRCH] Search failed in bpgetstat() function or bpsetstat().

[EADDRINUSE] Branchpoint already opened (bpopen() function).

The bpget() function may also fail and set errno for any of the errors, specified for the ***tsleep (9)*** and ***copyout (9)*** functions.

The bpopen() and bpput() functions may also fail and set errno for any of the errors, specified for the ***tsleep (9)*** and ***copyin (9)*** functions.

The bpgetstat() function may also fail and set errno for any of the errors, specified for the ***copyin (9)*** and ***copyout (9)*** functions.

The bpsetstat() function may also fail and set errno for any of the errors, specified for the ***copyin (9)*** function.

bpcall – kernel interface to branch point kernel module.

```
#include <branchpoint.h>
#define bpcall(args) branchpoint(NULL, (args))
```

Function bpcall() provides interface to branch point kernel module for other parts of OS kernel (for example, user CAMAC interrupt handler, see 3.2). bpcall() performs generic system call to branch point with argument of the struct branchpoint_args type (defined in the *branchpoint.h* header file as well as BPSYS number), pointed by args, which must be filled correspondingly. After return caller can revise contents of args for requested information.

Function bpcall() returns 0 on success, or the following values, if error occurred:

[EAGAIN] Free space in buffer not available or branch point in BPSTOP state.

[EPIPE] BPPUT operation failed due to branch point not opened for BPGET by anybody.

[EACCES] Branchpoint not opened.

[EMSGSIZE] Incorrect packet size specified for BPPUT operation.

[EINVAL] Invalid args supplied.

[EADDRINUSE] Branchpoint already opened.

[EOPNOTSUPP] Wrong sub–function in BPSYS system call specified.

Work or service module can be connected to branch point as:

input stream – BPPUT operations will be performed,

output stream – BPGET operations will be performed.

Kernel module (only one per branch point) can be connected only as input stream.

Each connected stream can be in the one of a three states:

BPRUN – packets will be put in / get from branch point buffer;

BPSTOP – process will be blocked (***tsleep (9)***), kernel module obtains [EAGAIN], when try to put in / get from branch point buffer, until state will be changed;

BPDISCARD, **input stream** – packets will not be putted in the branch point buffer, but call returns success;

BPDISCARD, **output stream** – packets will not be getted from the branch point buffer (identical to situation, when all available buffered packets already read).

3.7 Control modules

Current implementation provides at least following generic (setup/RUN independent) control modules: supervisor, ***bpgetstat(1)***, ***bpsetstat(1)***, ***bpclear(1)***, ***alarm(1)***, ***load(1)***, ***unload(1)***.

3.7.1 Supervisor

In the current implementation the **supervisor** is a control module (named *sv*), because of it does nothing with packet streams due to lack of control packets.
`sv [-q] [-e] [-p{-|<pidfile>}] [-c{-|<conffile>}]`

In the such synopsis form the *sv* opens dialogue with operator in the own X11 window, opened by *XForms* package and waits for operators command input. After obtaining such input *sv* performs corresponding action by ***make(1)***'ing some target(s) from default configuration file *sv.conf*.

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

`-q` Be quiet in dialogues with operator, use compiled–in defaults as much as possible (see all compiled–in defaults in *sv*'s *Makefile*).

`-e` Start in "expert mode", due to more details available for operator control.

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

`-c<conffile>` Read configuration from file named `<conffile>` instead of default. `-c-` means use compiled–in default for `<conffile>`.

The *sv* exits 0 on success, and > 0 on error. The *sv* ignores `SIGHUP`, `SIGINT` and `SIGQUIT` signals. For user termination in accuracy manner `SIGTERM` signal must be used. In the current implementation a supervisor configuration file ***sv.conf(5)*** is a makefile.

3.7.2 Syslog graphics presenter

In the current implementation a syslog graphics (X11) presenter is the *alarm* control module.
`alarm [-a] [-d{-|<display>}] [-p{-|<pidfile>}]`

In the such synopsis form the *alarm* reads standard input and displays its contents (strings contains "`Error`" and "`Warning`" substrings are highlighted) in the own X11 window, opened by *XForms* package on display, specified by:

- environment variable `DISPLAY`,
- command line option `-d` (see below),
- compiled–in default "`:0.0`"

(in the such order). So the *alarm* can be used as a graphics presenter for the ***syslogd(8)*** output (see example below).

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

`-a` Produce alarm beeps for highlighted strings by the ***speaker(4)*** facility.

-d<display> Use X11 display specification <display> as output display. -d- means use compiled–in default for <display>.

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

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

The *alarm* ignores SIGHUP, SIGINT and SIGQUIT signals. For user termination in accuracy manner SIGTERM signal must be used. The SIGPIPE signal also catched for accurate termination.

For example, the ***syslog.conf(5)*** file can contains the following entries:

```
local0.*            /var/log/qdpblog
local0.*;kern.*     |exec /usr/local/qdpb/bin/alarm -a -dlocalhost:0.0
#!handler
kern.*              /var/log/qdpblog
```

3.8 Event merger implementation

In the current implementation event merger is a loadable kernel module[5] of syscall type and must be loaded at syscall offset EMSYS (defined in the *eventmerger.h* header file). It provides access to its own input packet channels and output packet stream(s) for so called work and service modules – utilities for packet streams processing in the *qdpb* system. It implements user context interface described below.

emopen, emclose, emget, emput, emgetstat, emsetstat, emclear – user context interface to event merger kernel module.

```
#include <emio.h>
int emopen(int flag, struct emtbl *table)
int emclose(void)
int emget(void *buf)
int emput(const void *buf, int len)
int emgetstat(struct emargs *emargs)
int emsetstat(const struct emargs *emargs)
int emclear(void)
```

The emopen() function opens event merger for caller process with options, defined by flag argument. Options can constructed by OR'ing ("|") one of the following possible operations, defined in the *emio.h* header file:

EM_GET – open for get packets,

EM_PUT – open for put packets;

and one of the following possible states:

EM_RUN – open at run state,

EM_STOP – open at stop state,

EM_DISCARD – open at discard state.

[5]Implementation in a very preliminary stage.

Second argument table have meaning only for EM_PUT operation. It is a pointer to array of emtbl structures, defined in the *eventmerger.h* header file as follows:

```
struct emtbl {
        u_short in_type;          /* from input packet type ... */
        u_short out_type;         /* ... produce output packet type */
        u_char order;             /* piece ... */
        u_char number;            /* ... from total pieces */
};
```

Array need to be properly terminated (the table.order must be zero for last member), and may contains no more than EMMAXCORR (#define'd in the *eventmerger.h* header file) members including terminator. struct emtbl declares input emtbl.in_type and output emtbl.out_type packet types correspondence (see also 2.2). emtbl.order is a number, defines position of input packet's body in the output one. Valid range for it is from 1 to emtbl.number. emtbl.number is a total number of different input packet types, needed to produce such output packet. Valid range for it is from 1 to EMMAXCHAN (#define'd in the *eventmerger.h* header file).

The emclose() function closes event merger for caller process.

The emget() function gets one packet from event merger into destination, pointed by buf, and returns length of obtained packet.

The emput() function puts one packet with length len from source, pointed by buf, into event merger.

The emgetstat() function gets event merger status into emargs structure, defined in the *eventmerger.h* header file as follows:

```
struct emargs {
        pid_t p_pid;
        int oper;
        int stat;
        struct emtbl table[EMMAXCORR];
};
```

Before call emargs must contains valid (positive) p_pid. After call if oper == EM_PUT all fields will be filled and valid, if oper == EM_GET – all exclude table.

The emsetstat() function sets event merger status from filled by user source, pointed by emargs. Before call emargs must contains valid stat (EM_RUN, EM_STOP, or EM_DISCARD) and p_pid (nonnegative) fields. Zero p_pid means sets status for each registered process with user supplied oper (valid are EM_GET, EM_PUT, or its OR'ed combination means all registered processes). Status of process with (oper & EM_PUT) != 0 can not be changed individually (so for such oper only zero p_pid is valid).

The emclear() function clears output stream(s) of event merger.

The emget() returns 0 at EOF condition, all other functions returns 0 if successful. Otherwise −1 returned and global variable errno is set to indicate the error.

[EAGAIN] Free space in event merger not available (emput() function).

[EACCES] Eventmerger not opened (emget(), emput(), and emclose() functions).

[EMSGSIZE] Incorrect packet size (too short, too long, or not equal to header.len field value) specified in emput() function.

[EPIPE] EM_PUT operation failed due to event merger not opened for EM_GET by anybody (emput() function).

[EINVAL] Invalid state requested or invalid emtbl specified in emopen() function; invalid emargs specified in emgetstat() or emsetstat() functions; invalid header.id, header.len (too short or too long), or header.crc fields in supplied packet (emput() function).

[ENOBUFS] Space in table of registered process exceeded (emopen() function).

[ESRCH] Search failed in emgetstat() or emsetstat() functions.

[EADDRINUSE] Eventmerger already opened (emopen() function).

[EDOM] Value(s) of supplied emtbl.order member(s) is (are) invalid (out of range, more than emtbl.number, the same with previous call), two or more emtbl.in_type members are equals, two or more emtbl.out_type members are equals in emopen() function; type of proposed by emput() function input packet was not configured for such process.

[ERANGE] Value(s) of supplied emtbl.number member(s) is (are) invalid (out of range, different for different emtbl's or with previous call) in emopen() function.

[ENODEV] Value(s) of supplied emtbl.out_type member(s) other than already configured (emopen() function); not enough input packet channels opened to produce output packet (emput() function).

The emget() function may also fail and set errno for any of the errors, specified for the ***tsleep(9)*** and ***copyout(9)*** functions.

The emopen() and emput() functions may also fail and set errno for any of the errors, specified for the ***tsleep(9)*** and ***copyin(9)*** functions.

The emgetstat() function may also fail and set errno for any of the errors, specified for the ***copyin(9)*** and ***copyout(9)*** functions.

The emsetstat() function may also fail and set errno for any of the errors, specified for the ***copyin(9)*** function.

Work or service module can be connected to event merger as:

input stream – EMPUT operations will be performed,

output stream – EMGET operations will be performed.

All connected input stream(s) and each connected output stream(s) can be in the one of a three states:

EMRUN – packets will be put in / get from event merger;

EMSTOP – process will be blocked (***tsleep(9)***), when try to put in / get from event merger, until state will be changed;

EMDISCARD, **input stream** – packets will not be putted in the event merger, but call returns success;

EMDISCARD, **output stream** – packets will not be getted from the event merger (identical to situation, when all available packets already read).

4 Software dependencies and portability

The *qdpb* system currently uses the following third–party software:

camac2 package – CAMAC subsystem [5] implementation (see also 2.2).

netpipes package – ready implementation of some service modules (see also 3.5).

XForms package – GUI base for the supervisor ***sv (1)*** control module (can be replaced by *Xaw Toolkit* or *Open Motif Toolkit*, see below).

X Athena Widget (Xaw) Toolkit package – GUI base for some control modules (***sv (1)***, ***alarm (1)***, etc.).

Open Motif Toolkit package – GUI base for some control modules (***sv (1)***, ***alarm (1)***, etc.).

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

All *qdpb* code written on 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.

Packet bodies contents always produced in the little endian byte order.

All code for kernel context organized as devices, pseudo–devices, and loadable kernel modules and implements system calls and hardware interrupt handlers – all are typical kernel objects at modern UNIX–like OS's.

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 *qdpb* system source distribution contains *Makefile.def*, which initializes variables, used at *qdpb* system making, to default values. It `.include`'d by each *Makefile* in the *qdpb* source tree.

In the current implementation the *qdpb* system based on the FreeBSD operating system (4.x version at present time). *qdpb* sources are distributed as archive `qdpb-<yyyymmdd>.tar.gz`[6] (please contact `isupov@moonhe.jinr.ru`).

Acknowledgements

Authors have a pleasure to thank to V.G.Olshevsky and S.G.Reznikov for numerous fruitful discussions, essential help in testing, and some code contribution.

References

[1] Maurice J. Bach. **The design of the UNIX operating system.** Prentice–Hall Corp., New Jersey, 1986.

[2] U. Vahalia. **UNIX internals: the new frontiers.** Prentice–Hall Corp., New Jersey, 1996.

[6]<yyyymmdd> means some actual year, month, and day numbers.

[3] Olshevsky V.G., Pomyakushin V.Yu. **Use of UNIX on the Controlling Computer of the MUSPIN Setup.** Communication of JINR P10–94–416 (in Russian), Dubna, 1994.

[4] Gritsaj K.I., Olshevsky V.G. **Software Package for Work with CAMAC in Operating System FreeBSD.** Communication of JINR P10–98–163 (in Russian), Dubna, 1998.

[5] Gritsaj K.I. Private communications.

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

[7] Antyukhov V.A. et al. **Digital CAMAC modules (issue XVIII).** Communication of JINR P10–90–589 (in Russian), p.20, Dubna, 1990.

[8] Ibid., p.16.

[9] Lebedev N.I. Private communications.

[10] Basilev S.N., Slepnev V.M., Shutova N.A. **CAMAC crate controller CCPC4 on base of full complected IBM PC.** In: Proc. of XVII Int. Symposium on Nuclear Electronics, Varna, Sep. 15–21, 1997, p.192 (in Russian), Dubna, 1998.

Received by Publishing Department
on May 31, 2001.

Грицай К.И., Исупов А.Ю. E10-2001-116
Опыт реализации распределенной мобильной системы
сбора и обработки данных *qdpb*
(система обработки данных с точками ветвления)

Излагается проект распределенной мобильной (переносимой) системы сбора и обработки данных *qdpb*. Код, зависящий от аппаратного состава экспериментальной установки и характера собираемой информации, изолирован от общей части системы. Описывается реализация общей части системы.

Работа выполнена в Лаборатории высоких энергий и в Лаборатории ядерных проблем им. В.П.Джелепова ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна, 2001

Gritsai K.I., Isupov A.Yu. E10-2001-116
A Trial of Distributed Portable Data Acquisition
and Processing System Implementation:
the *qdpb* — Data Processing with Branchpoints

The project of a distributed portable data acquisition and processing system *qdpb* is issued. An experimental setup data and hardware dependent code is separated from generic part of the *qdpb* system. Generic part implementation is described.

The investigation has been performed at the Laboratory of High Energies, and at the Dzhelepov Laboratory of Nuclear Problems, JINR.

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

Макет Т.Е.Попеко

Подписано в печать 26.06.2001
Формат 60 × 90/16. Офсетная печать. Уч.-изд. л. 2,35
Тираж 300. Заказ 52717. Цена 2 р. 50 к.

Издательский отдел Объединенного института ядерных исследований
Дубна Московской области