The STAR DAQ Event Pool Interface and Raw Data Readers

Date: June 18. 2001
Version: 1.0BETA
Author: tonko@bnl.gov

Introduction

This document describes the software package which enables STAR users to access DAQ data produced online (class evpReader). The package also includes (or it will) raw data unpackers for all known detectors.

The package is distributed as a binary library file and a set of include files. The current supported systems are Linux and Solaris but the source is available and can easily be adapted to any Unix-like OS.

The package is a BETA version and doesn't have all the detectors in (i.e. it is lacking Level 3, RICH etc unpackers)! There may be bugs so let me know.

Package Organization

The package resides on the machine called **daqman.star.bnl.gov** in the top-level directory called /RTS. This directory is NFS-exported ReadOnly to the whole *.star.bnl.gov* domain. Locating the package in an AFS directory is foreseen but not currently supported.

Libraries:

`/RTS/lib/SUN/libevp.a`	Solaris, compiled for Ultra SPARCs
`/RTS/lib/LINUX/libevp.a`	386 Linux, compiled on RH 6.2

Include files:

`/RTS/include/EVP/evpReader.hh`
(which itself includes a bunch of other files in the `/RTS/include` tree.)

Sources:

`/RTS/src/EVP_READER` is the directory which has sources as well as a full fledged example in the file `special.C` using the `Makefile.special` makefile.

Users are welcome to look in the include file and the* `special.C` *example as well as the makefiles.!

Functionality

The package supports two distinct functions: getting the data event at a time and unpacking the raw data file to a "humanly" understandable simple format.

The input data source can be one of these three:

a) live Event Pool feed (the freshest event)
b) cached Event Pool data
c) standard raw DAQ data file

Option a) gets the newest event from the DAQ event stream which means that the caller may have to wait for an event to come through either because there aren't so many events or because there is no run in progress at this moment.

The Event Pool data resides on physical disks on the machine called **evp.star.bnl.gov** which are NFS exported to all of *.star.bnl.gov*. Each run is represented as a directory with that run's number as the name (i.e. "10234110") while each event is represented by a file inside that directory named by numbers according to the local sequence (i.e. "1","2",…"2331"). The sequence is "local" in the sense that the name is generated as the event arrives at the Event Pool and not by the event sequence coming from i.e. Trigger or such. This in turn means that there are no "holes" in the event naming sequence.

The Event Pool disks on **evp.star.bnl.gov** are additionally organized into disk volumes where the volumes are small letters starting from "a". An actual path to a specific event would look like this: `evp.star.bnl.gov:/a/12345678/23`.

In cases a) and b) above the client must be able to NFS mount the evp.star.bnl.gov volumes!

In the case c) above the user can use any raw DAQ data file which is readable i.e. locally on his disks, directly from DAQ's buffers (i.e. via *bufferbox*) etc.

The events are gathered one by one and made available to the data unpackers called "xxxReaders" where "xxx" is a detector name i.e. "tpc", "svt" etc. These detector readers unpack the raw data structures into a reasonable internal representation. I.e. the TPC data is made available to the final user as `tpc.adc[24][45][182][512]` i.e. in the natural detector representation of sectors, rows, pads, timebins. (*this example is not entirely correct but gives the right flavor!*).

Each reader has a statically allocated structure to store the data and thus may occupy considerable memory i.e. in the TPC case the reader needs about 300 MB. If, however, the user is not interested in the TPC at all she/he may just omit the call to the `tpcReader` unpacker function.

Use

class evpReader

The `class evpReader` (declared in `/RTS/include/EVP/evpReader.hh`) is responsible for opening the stream and getting an event from that stream. Please look at the class declaration in the include file above for details about the member variables as well as usage in the file `/RTS/src/EVP_READER/special.C`!

The "open" call is issued via the constructor which takes a `char *` as the only argument. There is *no* default constructor!

```
class evpReader *evp = new evpReader("yada") ;
class evpReader *evplive = new evpReader(NULL) ;
```

In the first example the reader will open the file system object pointed to by "yada" and it will automatically determine is it a file or a directory. In the second example the constructor will connect to the live, online event feed.

The constructor is very lightweight – it just allocates a few kB and proceeds with the initial setup.

Once the constructor returns the user *must* check the member variable `status` for success before proceeding. This variable will be non zero if the file/directory pointed to doesn't exist or is not readable or in, in case of the live feed, the corresponding daemon is down in which case please contact a DAQ person. In case of an error the only action possible is to call the destructor!

If the `evp->status` variable is zero (status is OK) the user may proceed and attempt to get an event via the class method `get` (how original, ☺) i.e.

```
for(;;) { // loop forever
     char *mem = evp->get(0,EVP_TYPE_ANY) ;
```

The get method takes exactly two integer arguments and returns a character pointer. The first argument should *always* be 0 (for now) while the second argument specifies the type of requested event. The type can be any ORed combination of the following defines:

EVP_TYPE_ZERO	just get a Token 0 event (i.e. the pedestals/RMS)
EVP_TYPE_PHYS	get a "physics" type (Trigger Command 4!) exclusivelly
EVP_TYPE_SPECIAL	get any special event type (Trigger Command != 4!)
EVP_TYPE_ANY	get any event that comes along (this is an OR of all the above)

Once the get call returns the user *must* check for a non-NULL pointer. If the return is NULL the user *must* interrogate the `status` member variable to determine how to proceed (see below). If the return is non-NULL the event is in memory (at that location) and is ready to be used. In this case the status variable is meaningless and one should not inspect it!

If and *only if* the call returns a NULL pointer the `status` variable will be one of the following:

EVP_STAT_OK There is no event yet – please repeat the call. The get method is designed to be non-blocking in case the caller is running inside a GUI thus it will always return as fast as possible if there is no event available from the Event Pool.

EVP_STAT_EOR Depending on the data source this can mean that DAQ is not currently running, you reached the end of file or this was the last event in the directory. Further action depends on the caller but the program would normally terminate unless the caller wishes to hang around waiting for a new run to start in the case of the live feed.

EVP_STAT_BAD An event was successfully received but something is wrong i.e. the event is corrupted in some way. One should generally reissue the get call.

EVP_STAT_CRIT Critical error occurred. Please let me know. No point in continuing whatsoever.

Once the get call returns a valid pointer one *should* first call the `datapReader` function i.e.

```
     char *datap = datapReader(mem) ;
```

Which checks the event header for consistency while the return character pointer (`char *datap` in the example) is used as the argument to *all* other readers.

Detector Readers

Each detector has a detector specific structure defined as (i.e. for the TPC):

```
static struct tpc tpc ;
```

and a reader function:

```
int ret = tpcReader(char *datap) ;
```

The reader function unpacks the data pointed to by the previous call to `datapReader` into the detector specific structure. The return value is one of:

0	no detector found in this event
<0	corrupted/unexpected data
>0	the size in bytes of the *whole* detector contribution including the misc. DAQ headers and such. The detector data is valid in this case *only*!

This call is usually blocking and may take some time to complete especially for detectors which have larger amounts of data (TPC, FTPC, SVT).

For the specifics of each detector please refer to the example code in `special.C` or to the detector specific include file `/RTS/include/EVP/xxxReader.h` for the comments.

Known Issues

Memory allocation on Linux systems is often broken especially in the early releases. If your system doesn't have enough "known" memory (physical + swap) the program will crash will Segmentation Violations (SIGSEGV) in the most unlikely places (i.e. inside a `malloc` call of an internal Linux subroutine i.e. `gethostbyname`?!) typically immediately on startup.

The only known fix (to me) is to increase the swap space on the machine just to "fool" the kernel and allow the program to load and run. The program won't use *that* much memory when it runs anyway.

Or, of course, don't use *all* the known readers but just the ones you are interested in.