Tips for Programming with GMRT Off-line Software

Sanjay Bhatnagar
N.C.R.A., Pune

March, 1997

1 Introduction

This document is intended to serve the dual purpose of a general description of the various tools developed for handling GMRT native data recording format (the LTA format), and a programming style guide for developing off-line data analysis applications.

The requirement of having a uniform user interface for all the off-line applications, imposes some mild programming standards. However, these are very mild and simple to be acceptable to all reasonable users.

The libraries for data handling have two layers. The bottom-most layer is written in C and can be used from C or FORTRAN applications. At the higher layer, some light weight C++ Objects have been built using the lower layer, which provides additional functionality without adding unreasonable complexity for the programmer.

We strongly recommend that programmers use the C++ layer for application programs.

2 Logical breakup of an application

Off-line applications are broken down into three major logical sections:

1. The user interface section (which uses the `cl` library[1])
2. The database section (which uses uses the `hlib` library or the C++ objects[2])
3. The data manipulation section (which uses the `glib` library)

[1] http://langur.ncra.tifr.res.in/~sanjay/Offline//ui
[2] http://langur.ncra.tifr.res.in/~sanjay/Offline//ltaobj

2.1 User Interface Section

In this section, the application interfaces to the standard off-line user interface via the `user interface libraries`[3] (also referred to as the `command-line library` or the `cl`-library) and determines the various parameters and their values which control the behavior of the application.

All applications using the `cl` library will expect parameters in the `<KeyWord>=<Val0,[Val1,Val2,..]>` format. These parameters can either be provided directly on the command-line, or can be set from an embedded interactive shell which the application will run. A complete description of the user interface can be found here[4].

A typical code in this section would be:

```
BeginCL(argc,argv);
  InteractiveCL(1);
  clgetFVal(.....);
  clgetSVal(.....);
     :
     :
EndCL();
```

`BeginCL()` is the entry point in the user interface library. `InteractiveCL()` is a signal to the library that by default, the application will run in the interactive mode. Various `clget...` calls are to query the user interface for the values of the keywords. `clgetSVal()` extracts the value of a keyword as a character string, `clgetFVal()` extracts the value(s) as a floating point number, etc. Simillary functions are written to extract values as `doubles` or `integers`. See the doumentation on the Application Programming Interface (API)[5] of user interface libraries for more details.

The `cl` library determines the list of keywords to which the application is sensitive, by recording all the queries made to it for the values of the various keywords. For this reason, one query per keyword *must* be made within the code section bracketed by calls to `BeginCL(...)` and `EndCL()`.

It is a good idea to extract all the information about the keywords within this section, and then proceed to the *Data base section*. However, there might be odd application which would query the user interface, outside the user interface section. A provision for this exist in the cl-library, but from the point of view of easily maintainable code, use of this is discouraged.

2.2 Database Section

In this section, the application mainly opens and sets up the visibility database using the database handling libraries. This is done by a call to `BeginHeader(char *InputFile, char *OutputFile)` for C applications. The first argument to `BeginHeader()` is the name of the input file and the second

[3] http://langur.ncra.tifr.res.in/~sanjay/Offline//uilib
[4] http://langur.ncra.tifr.res.in/~sanjay/Offline//ui
[5] http://langur.ncra.tifr.res.in/~sanjay/Offline//uilib

argument is the name of the output file to which binary output data (via `wt_data`) would be written. For C++ applications, this initialization is done by the constructor of the `LTAVIEW/LTAFMT`[6] objects. For C++ applications, the name of the input and output file is either supplied while instantiating the objects, or is set later by the `reset` method.

The application typically also extracts the the mappings of sampler to antenna names/numbers, of samplers to fft pipeline numbers and of sampler to baseline numbers in this section. These mappings are stored in structure called `fftmac`[7]. Routines for C and methods for C++ are provided to fill this structure from LTA database global header. While processing the data, tables inside this structure will allow fast data retrieval for any baseline and frequency channel from the data records.

The database initialization process will read the global header from the database. The C++ objects for data handling will also apply selections on the database based on the object name and also build internally, a table of offsets to the selected scans in the database for quick access to the data in the data-analysis section. The application then waits for request to read the input stream further. A request to read the database further is given by call to `int rd_data(char buf)` where `buf` points to a buffer of the size of the database record length. This reads the database in units of the database buffer length and returns the ID of the buffer. This ID could be `DATAREC` indicating that a data record was read, `NEWSCAN` indicating that a new scan was found, `UNKNOWNREC` indication a record of unknown type was found (usually indicating problem with the LTA database or a programming error), or `EOF`(=-1) indicating that the end-of-file was found.

Parameters related to the entire database can be extracted from the global header via the various `gget[ISFD]val(...)` calls while the latest scan header can be queried via the various `sget[ISFD]val(...)` calls (see the API[8]). The global header is available for the entire life of the application (unless explicitly destroyed) while only the latest scan header is available at any given time.

A typical code, with the database section looks like:

```
/* The user interface section */
BeginCL(argc,argv);
  InteractiveCL(1);
  clgetFVal(.....);
  clgetSVal(.....);
      :
      :
EndCL();

/* The database section */
BeginHeader();  /* This will get the input/output file */
 ggetsval(...); /* names from the cl library           */
```

[6]http://langur.ncra.tifr.res.in/~sanjay/Offline//ltaobj
[7]http://langur.ncra.tifr.res.in/~sanjay/Offline//gstruct
[8]http://langur.ncra.tifr.res.in/~sanjay/Offline//ltaobj

```
ggetival(...);
   :
   :
if (ggetival(``RECL'',&n)<0)
 ERROR;
   :
   :
while ((state=rd_data(buf) \!= -1))
  switch(state)
    {
      case NEWSCAN:
        /*Query the scan header here*/
         sgetSVal(...);
         sgetIVal(...);
            :
            :
      case DATAREC:
        /* This will be the data handling section*/
      case UNKNOWNREC:
        /* Error condition - in normal cases this will never be */
        /* reached */
    }
```

A C++ object called `LTAVIEW`[9] has been built around the `hlib` library with many additional facilities useful while analyzing the data. For e.g., using the C++ objects, applications can trivially handle more than one LTA file in a single application. The C++ object provides a "view" of a multi source LTA file as if only the user selected source scans are present in the database.

We believe that most applications should be written in C++ using this `LTAVIEW` C++ object. Minimal knowledge of the C++ syntax would be required to do this, in return of ease of programming. The most useful methods that will come handy when using `LTAVIEW` objects are the methods `avgb`, `avgn` and the inbuilt selection based on object names. Both these methods will behave exactly like the `rd_data` call of `hlib` except that `avgb` will return raw time averaged data buffer while `avgn` will return time averaged data buffer with each baseline normalized with the geometric mean of the two self correlations associated with the given cross correlation. Both these methods are sensitive to source selection and scan boundaries.

For e.g., a typical C++ code, which works on time averaged data will look like:

```
int i;
float TInteg;
LTAVIEW db; /* Instantiate the LTAVIEW object */
/* The user interface section */
```

[9] http://langur.ncra.tifr.res.in/~sanjay/Offline//ltaobj

```
i=1;
BeginCL(argc,argv);
  InteractiveCL(1);
  clgetFVal(.....);
  clgetSVal(.....);
  clgetFVal(``integ'',&TInteg,&i);
     :
     :
EndCL();

db.reset(InFile,OutFile,''.''); /* Initialize the LTAVIEW object  */
                                /* with the input,output files    */
                                /* The third argument is the name */
                                /* of the selected source.        */
                                /* ``.'' => ALL SOURCES           */
/* The database section */
 db.ggetval(...); /* Extract keyword values from global header */
 db.ggetval(...);
    :
    :
 if (db.ggetval(``RECL'',&n)<0) /* The named keyword was not found in
                                the global header */
  ERROR;
    :
    :
 while ((state=db.avgb(buf,TInteg) != EOF))
   switch(state)
     {
       case NEWSCAN:
         /*Query the scan header here*/
          db.sgetval(...);
             :
             :
       case DATAREC:
         /* Your data handling code...*/
       case UNKNOWNREC:
         /* Error condition - in normal cases this will never be */
         /* reached */
     }
```

As is clear from the above typical C++ code, the difference between C and C++ programming is negligible. As a function of time, we expect more and more astronomically useful methods which are dependent on the underlying data recording format, will become available (in this, or other derived C++ objects). It is therefore essential that programmers be familiar with use of these C++ objects.

2.3 Data analysis section

Most of the application level code for data analysis will be in this section. By the time the application reaches this stage, the data from the LTA Formatted binary file is available as a 2-D array of complex numbers as a function of baseline number and frequency channel. The `glib` library has some functions which can be of use in this section, and it is envisaged that as and when the need arises, programs will write astronomically useful functions and augment the `glib` library.

Since, in this section of the application, the data is available as a standard buffer which can be passed to routines written in any language, programmers can write the real data crunching code in the language of choice (often dictated by the requirements of efficiency and availability of ready made code).