- The sort functions should wrap the system `qsort` call
- Some of the Fourier transform functions should wrap the Fastest Fourier Transform in the West library (FFTW):

  `www.fftw.org`
- The FITS functions should wrap the CFITSIO library:

  `heasarc.gsfc.nasa.gov/docs/software/fitsio`
- Many of the astronomy routines will wrap the StarLink Positional Astronomy libraries (SLALib):

  `star-www.rl.ac.uk/star/docs/sun67.htx/sun67.html`
- libTAI will be used for time-related functions:

  `http://cr.yp.to/libtai.html`

In addition, some of the functions were implemented in a limited fashion in the Pilot Project. Also, RHL has provided functional prototype code for the memory management, tracing and message logging functions (some of which need to be revised), and the data containers (doubly-linked lists, hashes, arrays).

1.2 Threads

Pan-STARRS does not have a strong requirement for multithreading capability. The memory management functions, defined below, must be written to be thread-safe.

2 System Utilities

2.1 Configuration

It is important to be able to access the version of the code in use. `psLibVersion` shall return the current version of PSLib in use, as determined from CVS tags.

```
const char *psLibVersion(void);
```

2.2 Initialization

PSLib shall be initialized by calling `psLibInit` by the user before use. This enables the library to perform the following tasks that are required before general use:

- Set the error codes for PSLib.
- Seed the random number generator.

- Read the `psTime` configuration file (`timeConfig`) and set up the appropriate `psTimeTable`s and predictions.

The prototype is:

```
void psLibInit(bool predictable, const char *timeConfig);
```

If `predictable` is `true`, then the random number generator shall be seeded with a value that does not vary (allowing bugs relying on random numbers to be reproduced); otherwise, the random number generator shall be seeded from `/dev/random` if it exists, otherwise from the system clock.

`timeConfig` specifies the filename of the configuration file for `psTime` metadata.

2.3 Memory Management

2.3.1 Introduction

PSLib needs a level of memory management placed between the operating system (`malloc`/`free`) and the high level routines (e.g. `psMetadataAlloc`). This layer is in addition to the possibility that specific heavily used data types may need their own special-purpose memory managers. However, since we require that all user-level objects be allocated via associated `Alloc/Free` functions, we will easily be able to implement such functionality without impacting the facilities described here.

2.3.2 Rationale

We wish to insert our own layer of memory management for a number of reasons:

- We wish to insulate ourselves from the details of the system-provided `malloc`. There is no guarantee that the goals of the system architect align with those of the PSLib or the IPP.
- We need at least a wrapper layer which handles failed memory requests without requiring the application programmer to check every attempted allocation.
- We need to provide a mechanism for tracking and fixing memory leaks. While it is possible to do this by linking with external libraries (e.g. Electric Fence), it is convenient to do so within the framework provided by PSLib.
- Similarly, we wish to provide convenient hooks to detect and diagnose memory corruption.
- While debugging complex scientific code, it is very useful to be able to trace a given data structure as it passes through the processing pipeline.
- There may be other features that we wish to add in the future (e.g. associating a type with every allocation).