UNIVERSITY OF HAWAII AT MĀNOA
Institute for Astrononmy

Pan-STARRS Project Management System

Pan-STARRS PS-1 Image Processing Pipeline Library
Supplementary Design Requirements

Coop. Agreement No.	**: FA9451-06-2-0338**
Prepared For	**: Pan-STARRS PMO**
Prepared By	**: Eugene Magnier, Paul Price, Robert Lupton, Joshua Hoblitt**
Document No.	**: PSDC-430-007-21**
Document Date	**: July 24, 2006**
Revision	**: 21**

DISTRIBUTION STATEMENT
Approved for Public Release – Distribution is Unlimited

Submitted By:

[Insert Signature Block of Authorized Developer Representative] Date

Approved By:

[Insert Signature Block of Customer Developer Representative] Date

Revision History

Revision Number	Release Date	Description
DR	2004 Mar 29	Draft
00	2004 Apr 1	First version, sent to MHPCC
01	2004 May 19	Extensive modifications, see Appendix B
02	2004 Jun 22	Incorporation of Bugzilla PRs (up to 69)
03	2004 Jul 06	
04	2004 Jul 13	See Appendix B for a change log.
05	2004 Aug 16	draft for start of cycle 3
06	2004 Aug 19	revision for cycle 3
07	2004 Sep 07	final for cycle 3
08	2004 Oct 12	draft for start of cycle 4
09	2004 Nov 15	final for cycle 4
10	2004 Nov 30	update for cycle 4
11	2005 Jan 21	draft for cycle 5
12	2005 Feb 09	final for cycle 5
13	2005 Mar 30	draft for cycle 6
14	2005 Apr 27	final for cycle 6
15	2005 Jun 15	draft for cycle 7
16	2005 Sep 13	final for cycle 8
17	2005 Oct 18	draft for cycle 9
18	2005 Dec 06	draft for cycle 10
19	2006 Feb 21	draft for cycle 11
20	2006 Apr 11	draft for cycle 12
21	2006 Jul 24	final for cycle 13

Referenced Documents

Internal Documents

Reference	Title
PSDC-230-001	PS-1 Design Reference Mission
PSDC-430-004	Pan-STARRS PS-1 IPP C Code Conventions
PSDC-430-005	Pan-STARRS PS-1 IPP Software Requirements Specification
PSDC-430-006	Pan-STARRS PS-1 IPP Algorithm Design Document
PSDC-430-011	Pan-STARRS PS-1 IPP System/Subsystem Design Description

External Documents

Reference	Title
Posix Standard	Open Group Based Specifications Issue 6, IEEE Std 1003.1, 2003

Contents

1 Introduction and policies

This document describes the Pan-STARRS Image Processing Pipeline (IPP) Toolkit Library, PSLib. Any large software project such as the IPP benefits from the existence of a library of basic software functions which can be used throughout the project to simplify programming tasks. Among the benefits are the ability to reuse code, simplification of the testing process, streamlining of the code, and the isolation and encapsulation of concepts which may be subject to change. The component functions of such a library should represent well-defined, concise operations which can be coded with only a modest number of lines. PSLib is a library of basic functions required by the IPP, and it includes many programming concepts which may be useful for other software projects, especially those which deal with astronomical data handling tasks.

PSLib consists of a collection of library function calls — the Application Programming Interfaces (APIs) — and the associated data structures. The capabilities provided by PSLib are grouped into the following areas:

- System Utilities
- Basic Data Collections
- Data Manipulation
- Astronomy-Specific Functions.

This list is sorted in a hierarchical order: the later entries depend on the functions and data types of the earlier entries.

The installed code base for PSLib consists of header files, the binary library code, `libpslib.a` and the shared-library equivalent, `libpslib.so` (or `libpslib.dylib` in the case of OS/X). Assuming these components have been installed into the library and search path, PSLib may be used within a program by including the line `#include <pslib.h>` into the C code and linking with `-lpslib.`

This document describes the data structures and details the functions calls. The specified data structures and functions follow the naming conventions detailed in the IPP Software Requirements Specification (PSDC-430-005). In particular, these coding conventions restrict the namespace used by the library functions by requiring that all globally visible symbols start with the two letters `ps`. Further namespace organization is achieved by encouraging functions to be named in the form psNounVerbPhrase, where Noun is the data type of principle relevance and VerbPhase describes the operation applied to that data type. For example, the function which copies an image (of type `psImage`) is called `psImageCopy()`.

1.1 External Libraries

It is anticipated that many of the functions specified in this document will be implemented through wrapping external libraries:

- Many of the matrix functions, some of the polynomial and some of the minimization functions functions should wrap the GNU Scientific Library (GSL):
  `www.gnu.org/software/gsl`;
- 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`

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 some of the data containers (doubly-linked lists, hashes, arrays).

1.2 Threads and Re-entrancy

Due to current developments in CPU architecture, we must assume that PSLib will be used in a threaded environment. However, coding the library to be thread-safe may have implications for the speed of the library and for the simplicity of use. We therefore make the following policies:

- The memory management and error stack functions, defined below (Sections 2.3.3 & 2.5), must be written to be thread-safe, since we cannot risk this crucial area being unstable. However, this can have a large impact on the efficiency of the code, and so we specify that this behaviour may be activated and deactivated dynamically. The default behaviour, however, will be thread-safety, since it is more important to err on the side of being safe rather than efficient.
- Re-entrant versions of system calls and external library functions should be used. We expect that these cases are sufficiently small that we are prepared to err on the side of caution.
- The practise of using `static` variable to achieve high efficiency (e.g., so that subsequent calls do not have to repeat a large memory allocation) should be kept to an absolute minimum. Where it has been justified (i.e., through code profiling), the `static` variable must be protected by a "mutex".
- Cross-thread synchronization for PSLib's fundamental datatypes (`psArray`, `psList`, etc.) is left to the end-user (although some convenience is provided — see below).

As a convenience to the user in achieving thread-safe operation, each of the data structures classified as a "collection" (i.e., `psList, psHash, psMetadata, psArray, psPixels, psVector, psBitSet`) and `psImage` shall contain a member, `void *lock`, which provides a place for the user to carry around a mutex or semaphore. This is provided so that the user doesn't have to pass around both the structure and a mutex, or wrap PSLib structures in their own thread-safe structures that contain a mutex. PSLib is not responsible for allocating, setting, checking or freeing the `lock` — these are entirely the responsibility of the user. PSLib provides only a place to hang it. Your mileage may vary.

If these policies become a burden on the processing speed, we can investigate alternative measures, such as defining specifically re-entrant versions of select speed-critical functions.

1.3 Angles

To maintain consistency throughout the library, angles shall be specified in radians. In a small number of cases which we expect will be used heavily (i.e., `psSphereOffset` and trigonometric operations on vectors), the unit may be specified, which provides convenience to the user.

1.4 C++ Compatibility

All PSLib "public" header files should comptable with `C++`. This primarily involves using `C` pre-processor directives to enable `C++`'s `extern "C"` linkage specification when a header file is being processed as part of a `C++` compilation.

An example of wrapping a header file with an `extern "C"` block:

```
#ifndef FOO_H
#define FOO_H 1

#ifdef __cplusplus
extern "C" {
#endif

...

#ifdef __cplusplus
}
#endif

#endif // FOO_H
```

Other coding cautions include avoiding trailing commas in `enum`s, protection of the poison `pragma`s used for the memory system (below), and avoiding the qualifier `restrict`.

1.5 Use of `restrict`

The `restrict` type qualifier in C99 indicates to the compiler that the memory pointed to by a particular pointer is not also pointed to by some other pointer. This allows the compiler to optimise the code, based on the fact that aliasing is not an issue. However, the compiler does not check the accuracy of the assumption on which the optimisation is made, which can lead to data corruption.

Due to the large number of cross-links in psLib (e.g., the metadata container keeps two pointers to the data), the assumption behind the use of `restrict` generally will not hold. Correspondingly, use of `restrict` should be kept to a minimum; it should only be employed where necessary, and where the assumption will hold. Restricts should also be avoided because they are incompatible with `C++`.

1.6 Return values

In some cases, we have defined functions that return a boolean value. It is intended that (unless otherwise specified), the function returns `true` if there was no error, and returns `false` in the event of an error.

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

Initialization of the library is necessary in the general case. `psLibInit` shall initialize those elements of the library that require initialization. This includes loading the time configuration (through `psTimeInitialize`). In addition, if the environment variable `PS_ALLOC_CHECK` (analagous to `MALLOC_CHECK_` under GNU libc 2.x), then a memory check (checking for all leaks, and corruption) shall be performed at exit, with any problems reported to the file specified by `PS_ALLOC_CHECK`.

`psLibFinalize` shall free all memory used by psLib.

```
void psLibInit(const char *timeConfig);
void psLibFinalize(void);
```

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 specific 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).

2.3.3 Memory Management

In the following sections, we specify the API set and define the appropriate data structures needed by the PSLib memory management system in order to meet the requirements specified by the desiderata listed above.

Within the PSLib memory management system, every allocated memory block which is provided to the user is bounded by two additional memory segments. The segment preceeding the user-memory contains data describing the allocated block, using the `psMemBlock` structure. The first and last elements of this structure are `void` pointers called `startblock` and `endblock`, which are assigned a special value, `P_PS_MEMMAGIC`. The segment following the user-memory block consists of a single `void` pointer, and is also assigned the special value of `P_PS_MEMMAGIC`. The element `userMemorySize` specifies the number of bytes allocated in the user block and allows the endpost to be found.

In practice, these bounding memory blocks mean that when a user requests N bytes of memory, the memory management system in fact allocates `N + sizeof(psMemBlock) + sizeof(void)` bytes, starting at a particular address, `ADDR`. It then fills in the first `sizeof(psMemBlock)` bytes with the data of the `psMemBlock` structure, and the last `sizeof(void)` bytes with `P_PS_MEMMAGIC`. It returns to the user the pointer corresponding to the address `ADDR + sizeof(psMemBlock)`. If the memory management system reallocates a block of memory, it must also allocate the additional space and fill in the boundary values. If the memory management system is given a specific pointer for some operation, it is able to find the corresponding `psMemBlock` by simply subtracting `sizeof(psMemBlock)` from the pointer address.

The purpose of the three boundary markers is to catch corruption and to act as an aid in low-level debugging. In the first case, memory over- and under-run errors are likely to overwrite the special values in either the leading or trailing boundaries. The typical situation is one where the coder mis-counts the range and either fills the data just before the start of the valid memory or just after the end of the valid memory. These actions are likely to alter the boundary-post values, which can be detected by the memory management system. In the second case, hexadecimal dumps of large blocks of memory are easier to examine if the value of `P_PS_MEMMAGIC` is chosen to catch the eye. A traditional value for `P_PS_MEMMAGIC` is `0xdeadbeef` which is also easily recognized in a dump of the memory table.

The structure `psMemBlock` specifies additional information maintained for each block of allocated memory, and is defined as follows:

```
typedef struct psMemBlock {
    const void* startblock;             ///< initialised to p_psMEMMAGIC
    struct psMemBlock* previousBlock;   ///< previous block in allocation list
    struct psMemBlock* nextBlock;       ///< next block allocation list
    psFreeFunc freeFunc;                ///< deallocator.  If NULL, use generic deallocation.
    size_t  userMemorySize;             ///< the size of the user-portion of the memory block
    const psMemId id;                   ///< a unique ID for this allocation
    const char* file;                   ///< set from __FILE__ in e.g. p_psAlloc
```

```
    const unsigned int lineno;          ///< set from __LINE__ in e.g. p_psAlloc
    pthread_mutex_t refCounterMutex;    ///< mutex to ensure exclusive access to reference counter
    psReferenceCount refCounter;        ///< how many times pointer is referenced
    bool persistent;                    ///< marks if non-user persistent data like error stack, etc.
    const void* endblock;               ///< initialised to p_psMEMMAGIC
} psMemBlock;

typedef void (*psFreeFunc)(void* ptr);
typedef unsigned long psMemId;
typedef unsigned long psReferenceCount;
```

The PSLib memory management system must maintain the collection of allocated memory blocks. The entries `previousBlock` and `nextBlock` point to the previous and next memory blocks allocated by the memory management system (if they exist, or else `NULL`) and are used to scan through memory blocks as a linked list.

The element `freeFunc` specifies the deallocator associated with a specific block of memory. If this element is `NULL`, the basic deallocator (`psFree`) is used and the memory block must not be a rich data structure which requires additional freeing functionality (otherwise memory leaks will occur).

The element `id` in the structure is a sequential memory block ID. The memory management system must maintain an internal memory block ID counter from which a new ID may be supplied to each newly allocated block of memory and saved in the element `id`. This ID should also be the key to the memory block in the memory block table.

The two entries `file` and `lineno` are set to the line number and file at which the memory was originally allocated. This is most easily implemented by the use of the C preprocessor macros `__LINE__` and `__FILE__`. For this reason, we specify below that the basic memory managment functions be implemented as preprocessor macros which wrap the intrinsic C level functions.

The element `refCounterMutex` is a mutex used to limit access to the reference counter (below) to a single thread at a time.

The element `persistent` indicates if the memory is to be used by some non-user persistent data. This allows memory marked as `persistent` to be ignored when identifing memory leaks with `psMemCheckLeaks`. Private functions shall be provided to test and set the value of `persistent`:

```
void p_psMemSetPersistent(psPtr ptr, bool value);
bool p_psMemGetPersistent(psPtr ptr);
```

Sometimes system code will need to allocate complex structures (such as a `psMetadata`) that must all be marked as `persistent`. To save this code the trouble of tracking down every allocated pointer to mark each `persistent`, we define an additional private function, `p_psMemAllocatePersistent`, that sets the use of `persistent` within `psAlloc`. This allows us to call `p_psMemAllocatePersistent(true)`, allocate all our persistent memory, and then call `p_psMemAllocatePersistent(false)`. The initial setting shall be for allocated memory not to be marked `persistent`.

```
bool p_psMemAllocatePersistent(bool is_persistent);
```

The `psMemBlock` structure element `refCounter` is provided so APIs may cleanly manage multiple references to the same block of memory. As discussed below, the basic free function, `psFree`, is specified to free the memory block only if the reference counter is set to 1. See the discussion in section 2.3.9 for an example of the usage. Usage of this feature is strongly encouraged, but not enforced by the memory management system.

In order to trace double frees and other memory errors, the memory block reference is not automatically deleted when the assocated memory is deleted. Rather, the `psMemBlock` data and the endpost data are left behind. If `userMemorySize` is 0, then the memory block has been freed. This state must be enforced by `psFree`. This behavior, in which the associated `psMemBlock` is retained, is only provided if the code is compiled with the preprocessor variable `PS_MEM_DEBUG` defined.

2.3.3.1 Use of Persistent Memory

The `persistent` member of the `psMemBlock` is provided as a convenience to the end-user of psLib — it allows him to check for memory leaks in his own code, without going to the trouble of freeing memory allocated for psLib core functions. It also allows the use of psLib core functions (e.g., `psLogMsg, psTrace`) within the function that is called by `psMemCheckLeaks`, without the memory allocated by those psLib core functions being identified as a leak.

For these reasons, the `persistant` feature should only be used as necessary. In order to allow proper testing of psLib, all components that employ any `persistent` memory shall provide a function that frees all of its `persistent` memory (and only that persistent memory that belongs to that component), all of which should be called from `psLibCleanup`.

2.3.4 APIs for Allocating and Freeing

PSLib must provide the following APIs to create and destroy memory blocks:

```
psPtr psAlloc(size_t size);
psPtr psRealloc(psPtr ptr, size_t size);
void psFree(psPtr ptr);
```

From the user's perspective, the functions `psAlloc`, `psRealloc`, and `psFree` have identical semantics to the standard C library functions `malloc`, `realloc`, and `free`. In fact, these functions should be implemented as C preprocessor macros which call the following private functions:

```
psPtr p_psAlloc(size_t size, const char *filename, unsigned int lineno);
psPtr p_psRealloc(psPtr ptr, size_t size, const char *filename, unsigned int lineno);
void p_psFree(psPtr ptr, const char *filename, unsigned int lineno);
```

In these function calls, `size` is the number of bytes required to be allocated, `ptr` is the pointer to the allocated memory block, `file` is the file containing the calling function (set by `__FILE__`), and `line` is the calling function line number in the source-code file (set by `__LINE__`). Because the user should only see the preprocessor versions (ie, `psAlloc`), we do not distinguish between these two in the following discussion.

In order to enforce the use of the PSLib versions, the header file must take steps to ensure that code calling the functions `malloc`, `calloc`, `realloc`, or `free` will not compile. This may be achieved by defining preprocessor macros which mask these functions with invalid statements (e.g., `#define malloc(S) for`). In exceptional cases, such as the memory management system itself, programmers may choose to override this prohibition by defining the symbol `PS_ALLOW_MALLOC`. Application code will call `p_psAlloc`, `p_psRealloc`, or `p_psFree` via the macros defined above.

The above restrictions must apply strictly to the code within PSLib, and other dependent software, such as e.g., PSModules, may choose to enforce the restrictions as well. Software which employs PSLib should be allowed to optionally employ this enforcement or not, depending on which version of the global include file is chosen.

The functions `psAlloc` and `psRealloc` must never return a `NULL` pointer. If they are unable to provide the requested memory they must attempt to obtain the desired memory by calling the routine registered by `psMemExhaustedCallbackSet` (see §2.3.6), and if still unsuccessful, call `psAbort()`. The same behavior is true for constructors of rich structures, with names of the form `psFooAlloc`.

Note that we have not specified an equivalent of the `calloc` function. The `calloc` function provides two aspects which `malloc` originally did not: aligned memory and inialization. Neither of these are required: under POSIX, `malloc` is required to be aligned. Also, for all structures it is necessary to explicity define the initialization independently since a byte value of 0 is usually insufficient.

```
void psMemSetDeallocator(psPtr ptr, psFreeFunc freeFunc);
psFreeFunc psMemGetDeallocator(const psPtr ptr);
```

A free function handles any deallocation procedures of an object referred to by a pointer, excluding the freeing of the memory block of the pointer reference itself. If the pointer refers to an object that references other memory blocks this free function would insure the freeing of any encapsulated memory block references. For example, `psList` references a series of node objects of a linked list, so its free function would free these node objects and the node object's free function would free the data element references they may contain.

The fucntion `psMemSetDeallocator` is used to associated a free function to a memory block, while `psMemGetDeallocator` retrieves the last free function set. To remove a free function from a memory block, the `psMemSetDeallocator` should be invoked with `NULL`. If no free function is set, `psMemGetDeallocator` shall return `NULL`.

2.3.4.1 Negative allocations

Note that we have specified that the memory size is unsigned (`size_t`), so that we can address the full range of memory that the architecture will allow, and to match the behaviour of the system `malloc`. This creates the potential for problems if a negative value is inadvertently passed to `psAlloc` — it will be interpreted as a very large positive value. To guard against this, we specify that `psAlloc` must check that the allocation is less than `PS_MEM_LIMIT` (a preprocessor variable).

For array-like collections (specifically, `psArray`, `psPixels`, `psVector`, and `psImage`) we allow the user to refer to a negative index in the accessor (e.g., `psArrayGet`) to mean address from the end. Consequently, the number of elements in structures should be signed (in order to be able to access the full range of allocated values). It is the responsibility of these structure allocators (e.g., `psArrayAlloc`) to check that the requested number of elements is not negative (calling `psAbort` otherwise). All other allocators shall simply use `size_t` where the number of elements is needed (saving the trouble of checking before passing to psAlloc).

2.3.5 Callback Routines

The PSLib memory management system uses callback functions to handle certain errors and trace conditions. The callbacks are registered by the programmer and called by the basic memory management functions if needed. The four callbacks currently defined are called in the following situations:

- when insufficient memory is available (`psMemExhaustedCallback`)
- when a memory block is found to be corrupted (`psMemProblemCallback`)

- when a specified memory ID is allocated (`psMemAllocateCallback`)
- when a specified memory ID is freed (`psMemFreeCallback`)

The callback functions are defined in terms of specific callback types, specified below. The callbacks are set using functions with names of the form `psCallbackSet`. In all cases, the `Set` routine takes a pointer to the desired callback function and returns a pointer to the one that was previously installed. The default functions for each of these callbacks is listed below:

- `psMemExhaustedCallbackDefault`
- `psMemProblemCallbackDefault`
- `psMemAllocCallbackDefault`
- `psMemFreeCallbackDefault`

and have the behavior of immediately returning `NULL`, as if the callback had been skipped. If the function pointer passed to the functions above is `NULL`, the callback must be reset to the default callback function. The named default callbacks may be used in within a debugger to catch these conditions by breaking when the functions are called. We discuss the use of each of the four callbacks below.

2.3.5.1 `psMemExhaustedCallback`

If not enough memory is available to satisfy a request by `psAlloc` or `psRealloc`, these functions attempt to find an alternative solution by calling the `psMemExhaustedCallback`, a function which may be set by the programmer in appropriate circumstances, rather than immediately fail. The typical use of such a feature may be when a program needs a large chunk of memory to do an operation, but the exact size is not critical. This feature gives the programmer the opportunity to make a smaller request and try again, limiting the size of the operating buffer. This callback has the following form:

```
typedef psPtr (*psMemExhaustedCallback)(size_t size);

psMemExhaustedCallback psMemExhaustedCallbackSet(psMemExhaustedCallback func);
```

The callback function is called with the attempted size and is expected to return a pointer to the allocated memory or `NULL`. Until the callback function is set with `psMemExhaustedCallbackSet`, or if this callback is set to NULL, `psAlloc` immediately calls `psAbort`.

2.3.5.2 `psMemProblemCallback`

At various occasions, the memory manager can check the state of the memory stack. If any of these checks discover that the memory stack is corrupted, the `psMemProblemCallback` is called. The callback has the following form:

```
typedef void (*psMemProblemCallback)(psMemBlock *ptr, const char *filename, unsigned int lineno);

psMemProblemCallback psMemProblemCallbackSet(psMemProblemCallback func);
```

This callback may be used to report the error and other status information. No return value is accepted, and no specific operations are expected. The callback is for informational purposes only. Where practical and efficient, the memory manager must call the routine registered using `psMemProblemCallbackSet` whenever a corrupted block of memory is discovered. For example, doubly-freed blocks can be detected by checking `psMemBlock.refCounter`.

2.3.5.3 psMemAllocCallback & psMemFreeCallback

Two private variables, `p_psMemAllocID` and `p_psMemFreeID`, can be used to trace the allocation and freeing of specific memory blocks. If the first (`p_psMemAllocID`) is set and a memory block with that ID is allocated, the corresponding callback is called just before memory is returned to the calling function. If the second (`p_psMemFreeID`) is set and the memory block with the ID is about to be freed, the corresponding callback is called just before the memory block is freed. These variables are internal and private to the memory manager and should be set using the following two functions:

```
psMemId psMemAllocCallbackSetID(psMemId id);
psMemId psMemFreeCallbackSetID(psMemId id);
```

The corresponding callback functions have the following form:

```
typedef psMemId (*psMemAllocCallback)(const psMemBlock *ptr);
typedef psMemId (*psMemFreeCallback)(const psMemBlock *ptr);
```

and are set with the following functions:

```
psMemAllocCallback psMemAllocCallbackSet(psMemAllocCallback func);
psMemFreeCallback psMemFreeCallbackSet(psMemFreeCallback func);
psMemId psMemGetId(void);
```

The callback functions are called with a pointer to the corresponding memory block. The routines `psMemFreeCallbackIDSet` and `psMemAllocCallbackIDSet` accept the desired ID value and return the old value to the user. The return values of the handlers installed by `psMemAllocCallbackSet` and `psMemFreeCallbackSet` are used to increment the values of `p_psMemAllocID` and `p_psMemFreeID` respectively. For example, a return value of `0` implies that the value is unchanged; if the value is `2` the callback will be called again when the memory ID counter has increased by two. This functionality may be useful to check, for example, every 100th block allocated. The function `psMemGetId` returns the next identification number to be assigned to a memory block. This function can be used to guide the choice of ID set with the functions above.

2.3.6 Memory Tracing and Corruption Checks

The PSLib memory management system includes features to facilitate tracing the memory allocation and freeing process and to debug memory errors in the calling code. The types and function prototypes for this part of the memory API are shown below.

```
int psMemCheckLeaks(psMemId id0, psMemBlock ***array, FILE *fd, bool persistence);
int psMemCheckCorruption(bool abort_on_error);
```

The routine psMemCheckLeaks may be used to check for memory leaks. The return value is the number of blocks that have been allocated but not freed. Only blocks with psMemBlock.id greater than id0 are checked; this allows the user to ignore blocks allocated by initialization routines.

If the argument array is non-NULL, then **array is set to an array of pointers to psMemBlock when the function returns. These pointers represent the blocks which have been allocated but not freed. It is the caller's responsibility to free this array with psFree. Also note that **array should be NULL (or not point to allocated memory) upon entering the call or the corresponding memory reference will be lost.

If the argument fd is non-NULL, a one-line summary of each block that has been allocated but not freed is written to that file descriptor.

If persistence is false, then only those psMemBlocks not marked as persistent shall be considered as memory leaks. If persistence is true, then all psMemBlocks shall be considered as memory leaks.

The routine psMemCheckCorruption checks the entire heap for corruption, calling the routine registered with psMemProblemCallbackSet for each block detected as being corrupted. The return value is the number of corrupted blocks detected. If the argument abort_on_error is true, psMemCheckCorruption must call psAbort as soon as memory corruption is detected.

2.3.7 Reference Counting

As mentioned above, the memory management system provides the refCounter element in psMemBlock to allow for the management of multiple references to the same block of memory. External APIs which make use of this structure must increment the counter for every additional reference to an allocated memory block, and decrement it when those references are removed. The memory management routines respect the use of the refCounter field: psFree will not free a block for which refCounter is not 1, but shall decrement the refCounter, and psAlloc will initialize the field to 1. However, these functions do not (and cannot practically) enforce the use of the counters; this is a requirement of external APIs which intend to use the feature.

Several APIs are provided to manage the reference counters. These APIs are:

```
psReferenceCount psMemGetRefCounter(const psPtr ptr);
psPtr psMemIncrRefCounter(const psPtr ptr);
psPtr psMemDecrRefCounter(psPtr ptr);
```

The functions all take a pointer to the start of a user block of memory. The first simply returns the value of the reference counter. If vptr is NULL, this function must return a value of NULL. The next two functions increment or decrement the reference counter, returning the pointer which was passed in. These functions must validate the memory pointer by determining the corresponding psMemBlock.id and checking for consistency in the internal memory block table (the table pointer for psMemBlock.id must be in the valid range and must correspond to the address of the psMemBlock).

2.3.8 Thread safety

Locking a mutex is an expensive operation that can dramatically impact the efficiency of a program for the worse. When the user is not concerned with multiple threads, this is a needless waste, and so we specify that the thread safety in the memory management system may be deactivated (and activated) dynamically.

```
bool psMemSetThreadSafety(bool safe);
```

`psMemThreadSafety` shall turn on thread safety in the memory management functions if `safe` is `true`, and deactivate all mutex locking in the memory management functions if `safe` is `false`. The function shall return the previous value of the thread safety.

```
bool psMemGetThreadSafety(void);
```

`psMemGetThreadSafety` shall return the current state of thread safety in the memory management system.

Note that the default behaviour of the library shall be for the locking to be performed.

2.3.9 Relation of Memory Management to Structures

Within PSLib and throughout the Pan-STARRS project, we specify a variety of rich data structures. The IPP Software Requirements Specification states that structures should be defined with corresponding constructors and destructors. The destructors are private functions used only by the memory management system. Instances of, for example, `psSomeType` should be constructed using `psSomeTypeAlloc()` calls, and are destroyed using the basic `psFree` function, which calls `p_psSomeTypeFree()` to free the components of the structure, but leaves the task of freeing the structure to `psFree`. The allocator will allocate the required memory with `psAlloc` and increment the appropriate `refCounter`.

The existence of complicated structures which include pointers to other structures require that we lay out a rule regarding destructors and reference counters. Simply put, *the destructor for every structure should only free the structure if the* `refCounter == 1`*; otherwise, it decrements the reference counter and returns.*

2.3.10 Conventions adopted for pointers

Only pointers to memory allocated with the PSLib memory functions are compatible with the various PSLib container types (e.g., `psList, psMetadata`), because the functions working with the container types search for the attached `psMemBlock`. If a pointer to memory allocated with another memory system (e.g., the system `malloc`), or generated by offsetting from another pointer that was allocated with `psAlloc`, is used with PSLib, the PSLib functions would falsely determine that memory is corrupted, because of the missing `psMemBlock`.

To pilot our way through the potential confusion, instead of calling all pointers (of unspecified type) a "`void*`", we adopt a convention, both in this document and to be used in the source, of referring to a pointer that has a `psMemBlock` attached as a `psPtr`:

```
typedef void* psPtr;
```

For the same reason, we also adopt a convention of referring to a string that has a `psMemBlock` attached as a `psString`:

```
typedef char* psString;
```

That is, `psPtr` is used in cases where the function requires the use of a pointer of unspecified type allocated with `psAlloc`, and `psString` is used in cases where the function requires the use of a `char*` allocated with `psAlloc` (e.g., via `psStringCopy` or `psStringNCopy`).

2.3.11 Strings

The use of strings within the PSLib memory management system requires that they be allocated with `psAlloc`. This means that substrings cannot be placed on containers such as `psArray` or `psMetadata`, since these check for the existence of the attached `psMemBlock`. To get around this, we specify functions that copy a string into `psAlloc`-ed memory:

```
psString psStringCopy(const char *string);
psString psStringNCopy(const char *string, unsigned int nChar);
```

`psStringCopy` shall perform a deep copy of the `string` into `psAlloc`-ed memory. `psStringNCopy` shall do the same, up to a maximum of `nChar` characters. Both copy functions shall "pass through" `NULL` values without generating an error.

We also specify two useful functions:

```
ssize_t psStringAppend(char **dest, const char *format, ...)
ssize_t psStringPrepend(char **dest, const char *format, ...)
```

`psStringAppend` shall append, according to the `format` values into the `dest` string. `dest` shall be allocated if `NULL`, and the length of the new string (excluding the terminator) shall be returned. `psStringPrepend` shall do similarly, except it shall prepend to the `dest` string.

Other string manipulation functions are listed below.

```
psList *psStringSplit(const char *string, const char *splitters, bool multipleAreSignificant);
```

`psStringSplit` shall split the input `string` into a `psList` of `psStrings`. The `string` is split at any one of the characters in `splitters`. Split strings of zero length should not be included in the output list if `multipleAreSignificant` is `true`.

String whitespace from head and tail of string:

```
size_t psStringStrip(char *string);
```

Substitute a `key` with `replace` in the `input` string:

```
psString psStringSubstitute(psString input, const char *replace, const char *key);
```

2.3.12 Fixed-Length Lines

We define the `psLine` structure to carry a pre-allocated block to store a character string. The interfaces are similar to `psString`, but allow for the container to be allocated initially upfront with a specified length. They are used to force a fixed-length line.

```
// structure to carry a dynamic string
typedef struct {
    long NLINE;                         // allocated length
    long Nline;                         // current length
    psString line;                // character string data
} psLine;
```

The following function allocates a line object of length Nline.

```
psLine *psLineAlloc(long Nline);
```

The following function initializes or re-initializes the line, setting the current length to zero and setting the string data values to 0. If the function is passed `NULL`, the function returns `false`.

```
bool psLineInit(psLine *line);
```

The following function appends a string to a line segment, returning `false` if the new segment would overflow the allocated line length.

```
bool psLineAdd(psLine *line, const char *format, ...);
```

2.3.13 Type information

We need to be able to specify two different sorts of types.

The first, for use with math structures (§4), defines a numerical type; e.g., short integer, double-precision floating point, etc:

```
typedef enum {
    PS_TYPE_S8      = 0x0101,           ///< Character
    PS_TYPE_S16     = 0x0102,           ///< Short integer
    PS_TYPE_S32     = 0x0104,           ///< Integer
    PS_TYPE_S64     = 0x0108,           ///< Long integer
    PS_TYPE_U8      = 0x0301,           ///< Unsigned character
    PS_TYPE_U16     = 0x0302,           ///< Unsigned short integer
    PS_TYPE_U32     = 0x0304,           ///< Unsigned integer
    PS_TYPE_U64     = 0x0308,           ///< Unsigned long integer
    PS_TYPE_F32     = 0x0404,           ///< Floating point
    PS_TYPE_F64     = 0x0408,           ///< Double-precision floating point
    PS_TYPE_C32     = 0x0808,           ///< Float complex
    PS_TYPE_C64     = 0x0810,           ///< Double complex
    PS_TYPE_BOOL    = 0x1301            ///< Boolean value
} psElemType;
```

The second, primarily for use with the metadata (§3.6), defines a data structure; e.g., list, array, FITS file:

```
typedef enum {                          ///< type of item.data is:
    PS_DATA_S8      = PS_TYPE_S8,       ///< psS8
    PS_DATA_S16     = PS_TYPE_S16,      ///< psS16
    PS_DATA_S32     = PS_TYPE_S32,      ///< psS32
    PS_DATA_S64     = PS_TYPE_S64,      ///< psS64
    PS_DATA_U8      = PS_TYPE_U8,       ///< psU8
    PS_DATA_U16     = PS_TYPE_U16,      ///< psU16
    PS_DATA_U32     = PS_TYPE_U32,      ///< psU32
    PS_DATA_U64     = PS_TYPE_U64,      ///< psU64
    PS_DATA_F32     = PS_TYPE_F32,      ///< psF32
    PS_DATA_F64     = PS_TYPE_F64,      ///< psF64
    PS_DATA_BOOL    = PS_TYPE_BOOL,     ///< psBool
```

```
    PS_DATA_STRING  = 0x10000,         ///< String (char *)
    PS_DATA_ARRAY,                     ///< psArray
    PS_DATA_BITSET,                    ///< psBitSet
    PS_DATA_CUBE,                      ///< psCube
    PS_DATA_FITS,                      ///< psFits
    PS_DATA_HASH,                      ///< psHash
    PS_DATA_HISTOGRAM,                 ///< psHistogram
    PS_DATA_IMAGE,                     ///< psImage
    PS_DATA_KERNEL,                    ///< psKernel
    PS_DATA_LINE,                      ///< psLine
    PS_DATA_LIST,                      ///< psList
    PS_DATA_LOOKUPTABLE,               ///< psLookupTable
    PS_DATA_METADATA,                  ///< psMetadata
    PS_DATA_METADATAITEM,              ///< psMetadataItem
    PS_DATA_MINIMIZATION,              ///< psMinimization
    PS_DATA_PIXELS,                    ///< psPixels
    PS_DATA_PLANE,                     ///< psPlane
    PS_DATA_PLANEDISTORT,              ///< psPlaneDistort
    PS_DATA_PLANETRANSFORM,            ///< psPlaneTransform
    PS_DATA_POLYNOMIAL1D,              ///< psPolynomial1D
    PS_DATA_POLYNOMIAL2D,              ///< psPolynomial2D
    PS_DATA_POLYNOMIAL3D,              ///< psPolynomial3D
    PS_DATA_POLYNOMIAL4D,              ///< psPolynomial4D
    PS_DATA_PROJECTION,                ///< psProjection
    PS_DATA_REGION,                    ///< psRegion
    PS_DATA_SCALAR,                    ///< psScalar
    PS_DATA_SPHERE,                    ///< psSphere
    PS_DATA_SPHEREROT,                 ///< psSphereRot
    PS_DATA_SPLINE1D,                  ///< psSpline1D
    PS_DATA_STATS,                     ///< psStats
    PS_DATA_TIME,                      ///< psTime
    PS_DATA_VECTOR,                    ///< psVector
    PS_DATA_UNKNOWN,                   ///< Other data of an unknown type
    PS_DATA_METADATA_MULTI             ///< Used internally for metadata; not a 'real' type
} psDataType;
```

Here we have included every type of structure used in PSLib that we expect will be frequently carried around in containers. If applicable, the value should corespond to the same type as represented in `psElemType`.

The other values are offset from these "primitive" types so that they can be identified easily. A macro, `PS_DATA_IS_PRIMITIVE(type)`, shall return `true` if the type is one of the numerical data types (S32, F32, F64, bool). In such a case, the data value is directly available from the metadata. Otherwise, a pointer to the data is available.

`PS_DATA_METADATA_MULTI` is used by the metadata, so the user should not use this type.

2.3.14 Type checking

Several of the collections contain data using a `void*` pointer. This makes it difficult to ensure that data coming off a collection is of a particular, desired type. Nevertheless, there is a way of identifying the type of a pointer pulled off a collection — the registered deallocator function stored in the `psMemBlock`. We specify functions below, one for each of the major types in psLib, that check the type on a particular pointer, returning `true` if the `ptr` matches the desired type, as determined from the registered deallocator function. These functions may be implemented as macros or inline functions if that is deemed convenient.

```
bool psMemCheckArray(psPtr ptr);
bool psMemCheckBitSet(psPtr ptr);
bool psMemCheckCube(psPtr ptr);
bool psMemCheckFits(psPtr ptr);
bool psMemCheckHash(psPtr ptr);
bool psMemCheckHistogram(psPtr ptr);
bool psMemCheckImage(psPtr ptr);
bool psMemCheckKernel(psPtr ptr);
bool psMemCheckList(psPtr ptr);
bool psMemCheckLookupTable(psPtr ptr);
bool psMemCheckMetadata(psPtr ptr);
bool psMemCheckMetadataItem(psPtr ptr);
bool psMemCheckMinimization(psPtr ptr);
bool psMemCheckPixels(psPtr ptr);
bool psMemCheckPlane(psPtr ptr);
bool psMemCheckPlaneDistort(psPtr ptr);
bool psMemCheckPlaneTransform(psPtr ptr);
bool psMemCheckPolynomial1D(psPtr ptr);
bool psMemCheckPolynomial2D(psPtr ptr);
bool psMemCheckPolynomial3D(psPtr ptr);
bool psMemCheckPolynomial4D(psPtr ptr);
bool psMemCheckProjection(psPtr ptr);
bool psMemCheckScalar(psPtr ptr);
bool psMemCheckSphere(psPtr ptr);
bool psMemCheckSphereRot(psPtr ptr);
bool psMemCheckSpline1D(psPtr ptr);
bool psMemCheckStats(psPtr ptr);
bool psMemCheckTime(psPtr ptr);
bool psMemCheckVector(psPtr ptr);
```

For user convenience, we also specify a one-stop shop which simply executes the appropriate `psMemCheckWhatever` function according to the `type`:

```
bool psMemCheckType(psDataType type, psPtr ptr);
```

The reason for having functions that check the type instead of returning the type is that the check is quick and the result is clean, while returning the type involves descending through a case statement.

2.4 Tracing and Logging

This section defines the Pan-STARRS Tracing and Logging APIs; the former refers to debug information that we wish to be able to turn on and off without recompiling (it will *not* be available in production code); the latter means information about the processing that must be collected and saved, even in the production system. We envision that extensive use will be made of `psTrace` throughout the Pan-STARRS code.

2.4.1 Tracing APIs

A code-tracing facility should allow the programmer to place messages in the code which, when called, will print some useful information about the containing block of code. Ideally, different messages may be specified to have different

levels (of severity or interest). For a given run of the program, the level of interest should be set to provide more or less feedback, depending on the needs of the programmer. In a typical situation, low-level messages would be placed generously throughout the code, indicating the flow of the program. Higher-level messages would be placed in a limited number of special locations, such as the start of major code segments or where a particularly unusual condition is met. Top-level messages would be placed in code triggered under serious error conditions. A normal run of the program would have the trace messages printed only for the top-level. If the user needs to dig deeper into the code, the trace level should be set lower, and the more detailed messages could be examined. In a case of a serious, poorly-understood problem with the code, the trace threshold would be placed to the bottom and the lowest-level step-by-step messages would be printed.

The PSLib tracing facility provides the above functionality, along with the ability to assign different trace levels to messages from different software components. Each trace message, when placed in the code, is assigned to be part of a specific tracing 'facility', defined in more detail below. The trace level for that specific message is also set when the message is placed. Each facility may have its trace level set independently. Thus, it is possible to request detailed trace output for one facility while minimizing the verbosity of the trace output from the rest of the program.

The trace facilities consist of a hierarchy of names. A trace facility is defined by a string consisting of words separated by dots, with a hierarchy equivalent to that of UNIX directory names. The top-level facility is simply `'.'` (one dot). The next level would be `'.A'`, followed by `.A.B`, and so on. The relationship is seen in two ways. First, a facility inherits the trace level of its parent unless explicitly specified. Second, the hierarchy is used to format the listing of the trace facilities so that they are easy to read. The first of these rules provides a mechanism to define the default trace levels for any facility even if it has not been registered explicitly since all named facilities are implicitly children of the top level facility (`.`). The second rule is simply an organizational technique to make the listing of facility information clear. In specifying a facility, the leading dot shall be optional, as a convenience to the user.

The API to place a trace message in the code, and simultaneously set its trace level and facility, is:

```
void psTrace(const char *facil, int level, const char *format,...);
void psTraceV(const char *facil, int level, const char *format, va_list ap);
```

where the `format` argument is a printf-style formatting code followed by possible arguments to that formatting statement, to be implemented using the `vprintf` functions. This command specifies the name of the facility to which the message belongs (`facil`), the trace level for this message in that facility (`level`) and the message itself. The `psTraceV` version of the command accepts a `va_list` argument list rather than a variable number of arguments.

The trace level for any facility may be set at any time with the following function:

```
int psTraceSetLevel(const char *facil, int level);
```

where `level` specifies the current trace level for the facility named by `facil`. The function returns the previous trace level for that facility. The currently defined trace level for a given facility may be determined by the function:

```
int psTraceGetLevel(const char *facil);
```

which returns the trace level of the named facility following the rules specified above. A specified trace message (identified by `psTrace`) must be printed if and only if `psTraceGetLevel(facil)` returns a value greater than or equal to the value of `level` for that message. That is, a larger number for the trace level corresponds to lower-level statements, and hence is more verbose.

PSLib includes a utility function for examining the current tracing levels of all facilities:

```
void psTracePrintLevels(void);
```

This function prints the hierarchy of trace facilities along with the current trace level for each facility. For example, a particular program may have a few facilities defined, along with their trace levels. A call to `psTracePrintLevels` may produce a listing which appears as:

```
.                       0
 .IPP                   0
 .IPP.debias            1
 .IPP.flatten           1
  .IPP.flatten.divide   2
  .IPP.object.findpeak  1
  .IPP.object.getsky    1
```

Considering the same program, the programmer might place a variety of trace messages throughout the `flatten` portion of the code with different types of messages, such as:

```
psTrace("IPP.flatten", 2, "starting flatten function\n");
psTrace("IPP.flatten", 0, "flat-field image is \%s\n", filename);
psTrace("IPP.flatten.divide", 2, "doing the divide\n");
psTrace("IPP.flatten.divide", 3, "trying the loop, i = \%d\n", i);
psTrace("IPP.flatten.divide", 1, "got an invalid pixel value (NaN) at \%d,\%d\n");
psTrace("IPP.flatten.divide", 2, "divide is done\n");
```

Under the trace levels set above, if the code actually reached each of these trace messages, the following messages would be printed:

```
flat-field image is foo.fits
  doing the divide
  got an invalid pixel value (NaN) at 500,20
  divide is done
```

while these two would not be printed because their facility level was too low:

```
  starting flatten function
   trying the loop, i = 0
   trying the loop, i = 1
   trying the loop, i = 2
...
```

The availability of the tracing facility at run-time, must be decided at compilation: If the C pre-processor macro `PS_NO_TRACE` is defined, all trace code must be replaced by empty space so that none of the code is compiled. This can be implemented via macro front-ends to private versions of the user APIs. In addition, a function `void psTraceReset(void)` will free memory used by `psTrace` functions, effectively resetting all trace levels to 0.

The trace may optionally be written to a file or other output destination with `psTraceSetDestination`:

```
bool psTraceSetDestination(int fd);
```

If the `fd` is zero, then tracing is disabled. Otherwise, the trace is sent to the specified file descriptor. As a convenience, the following are defined:

```
enum {
    PS_TRACE_TO_NONE = 0,               ///< turn off all traces
    PS_TRACE_TO_STDOUT = 1,             ///< trace to system's stdout
    PS_TRACE_TO_STDERR = 2,             ///< trace to system's stderr
};
```

This arrangement mirros the file descriptors for standard input, output and error. A call to `psTraceSetDestination` automatically closes an open file descriptor.

The corresponding function

```
int psTraceGetDestination(void);
```

returns the current trace destination file descriptor. If the destination has not been defined by the user, the descriptor for `stdout` is returned.

The trace output format is controlled with the function:

```
bool psTraceSetFormat(const char *format);
```

which expects a string consisting of the letters `H` (host), `L` (level), `M` (message), `N` (name), `F` (file:name), and `T` (time). The default is `THLNM`, which produces trace messages in the form:

```
YYYY-MM-DD hh:mm:ssZ | hostname | L | name | file:line
    The message goes here
    and is indented by 4 spaces.
```

where `YYYY`, `MM`, `DD`, `hh`, `mm`, and `ss` are the year, month (Jan is 01), day of the month, hours (0–23), minutes, and seconds when the trace message was received. Note that the timestamp is in ISO order, and that the timezone is GMT (hence the `Z`). The `hostname` is returned by `gethostname`, `L` is the numerical level. The other two fields, `name` and `msg`, are the facility name and the complete message provided to `psTrace`. The `msg` is placed on a new line (allowing the `name` to fill the rest of the previous line), with each line indented by 4 spaces. An example message is:

```
2004:02:24 20:14:18Z | alibaba.IfA.Hawaii.Edu | I | example.utils.helloWorld | example.c:20
    Hello world,
    it's me calling.
```

The possible order of the format entries is fixed and not determined by the order of the letters used in `psTraceSetFormat`. Selecting an output format with fewer than the complete set of 5 entries simply removes those entries from the output messages.

Specifying a `format` of `NULL` turns off logging (equivalent to calling `psTraceSetDestination(PS_TRACE_TO_NONE)`, whereas if the `format` is `""`, then the format reverts to the default.

2.4.2 Message Logging

Message logging is similar in some respects to tracing. Like trace messages, log messages are placed in the code at various locations to provide output describing the current state of the program. Like the PSLib trace facility, a good log facility should have the capability of associating each message with an importance or severity level, and at any point, the level for which messages are actually printed should be set in a flexible manner. Unlike trace messages, however, log messages are always part of the code and are available in the production version as well as in test versions.

The PSLib logging facility does not include the extensive facility levels which are provided by the trace facility. Less control over the granularity is needed for the log messages than for the trace messages.

A log message is placed in the code with the command:

```
void psLogMsg(const char *name, int level, const char *format, ...);
void psLogMsgV(const char *name, int level, const char *format, va_list ap);
```

where `name` is a word to describe the source of the message, `level` is the severity level of this message, and `format` is a printf-style formatting statement defining the actual message, and is followed by the arguments to the formatting code. The second form, `psLogMsgV` is an equivalent command which takes a `va_list` argument.

A log message may have any level specified in the range 0-9, though the first 4 levels are associated with symbolic names:

```
enum { PS_LOG_ABORT = 0, PS_LOG_ERROR, PS_LOG_WARN, PS_LOG_INFO };
```

At any time, the program may set the current log level, the level above which log messages are ignored, using the function:

```
int psLogSetLevel(int level);
```

This function returns the previous log level. A specific message invoked with `psLogMsg` is only printed if its value of `level` is less than the current value set by `psLogSetLevel`. The default log level is set to `PS_LOG_INFO`.

```
int psLogGetLevel();
```

This function returns the current log level.

Log messages are sent to the destination most recently set using:

```
bool psLogSetDestination(int fd);
```

If the `fd` is zero, then logging is disabled. Otherwise, the log is sent to the specified file descriptor. As a convenience, the following are defined:

```
enum {
    PS_LOG_TO_NONE = 0,                 ///< turn off logging
    PS_LOG_TO_STDOUT = 1,               ///< log to system's stdout
    PS_LOG_TO_STDERR = 2,               ///< log to system's stderr
};
```

This arrangement mirros the file descriptors for standard input, output and error. A call to `psLogSetDestination` automatically closes an open file descriptor.

The corresponding function

```
int psLogGetDestination();
```

returns the current log destination file descriptor. If the destination has not been defined by the user, the descriptor for `stdout` is returned.

The output format is controlled with the function:

```
bool psLogSetFormat(const char *format);
```

which expects a string consisting of the letters H (host), L (level), M (message), N (name), F (file:line), and T (time). The default is THLNM, which produces log messages in the form:

```
YYYY-MM-DD hh:mm:ssZ | hostname | L | name | file:line
    The message goes here
    and is indented by 4 spaces.
```

where YYYY, MM, DD, hh, mm, and ss are the year, month (Jan is 01), day of the month, hours (0–23), minutes, and seconds when the log message was received. Note that the timestamp is in ISO order, and that the timezone is GMT (hence the Z). The hostname is returned by gethostname, L is a character associated with the level (A, E, W, and I for PS_LOG_ABORT, PS_LOG_ERROR, PS_LOG_WARN, and PS_LOG_INFO respectively. Other levels are represented numerically (5 etc.). The other two fields, name and msg, are the arguments to psLogMsg. The msg is placed on a new line (allowing the name to fill the rest of the previous line), with each line indented by 4 spaces. An example message is:

```
2004:02:24 20:14:18Z | alibaba.IfA.Hawaii.Edu | I | example.utils.helloWorld
    Hello world,
    it's me calling.
```

The possible order of the format entries is fixed and not determined by the order of the letters used in psLogSetFormat. Selecting an output format with fewer than the complete set of 5 entries simply removes those entries from the output messages.

Specifying a format of NULL turns off logging (equivalent to calling psLogSetDestination(PS_LOG_TO_NONE), whereas if the format is "", then the format reverts to the default.

The following utility opens an output file descriptor for use by the trace and log facilities.

```
int psMessageDestination (const char *dest);
```

The destination string consists of a URL in the form protocol:location. The protocol value may be file, to send the log to a local file named by the value of location. Future expansion may allow the logger to send messages to an IP logger, with a protocol to be defined later. Three other special values are allowed for the dest parameter (without specifying a protocol): stderr and stdout, which return the file descriptors for stderr and stdout respectively, and none which returns the special descriptor to turn off logging.

2.5 Error Handling

Pan-STARRS errors shall be raised using a function, psError, with the caller supplying a component name and error message. It is desireable to be able to trace an error through the program so that the events that led to the error may be quickly and clearly identified. psError prints an error message and doesn't abort, but instead returns the error code.

```
psErrorCode p_psError(const char *filename,
                      unsigned int lineno,
                      const char *func,
                      psErrorCode code,
                      bool new,
                      const char *format, ...);
```

```
#define psError(code, new, format, ...) \
    p_psError(__FILE__, __LINE__, __func__, code, new, format, __VA_ARGS__)
```

psError is a macro definition that allows the filename, line number and function name to be inputted to a private function, p_psError. The code is an enumerated type which lists the possible *classes* of errors (e.g. PS_ERR_IO) that Pan-STARRS code can generate (see section 2.5.1). The new argument takes a boolean which, if true specifies that the error was set initially at this location, and if false specifies that an error was passed to this location. Raising new error should clear the error stack. The final required argument, format, is a printf-style format that is passed to psLogMsgV with code PS_LOG_ERROR, and component equal to the concatenation of the file name and the line number, separated by a colon. The result of a call to psError shall be to push an error onto a stack; this stack is cleared if new is true, or by a call to psErrorClear.

The errors on the error stack are defined as the following:

```
typedef struct {
    char *name;                         ///< category of code that caused the error
    psErrorCode code;                   ///< class of error (equivalent to errno)
    char *msg;                          ///< the message associated with the error
} psErr;
```

The last error reported is available from psErrorLast; if no errors are current, a non-NULL psErr shall be returned with code PS_ERR_NONE. Previous errors on the stack shall be returned by psErrorGet (a value of 0 passed to psErrorGet is equivalent to a call to psErrorLast). The error stack may be completely cleared with psErrorClear.

```
unsigned int psErrorGetStackSize(void);
const psErr *psErrorGet(int which);
const psErr *psErrorLast(void);
void psErrorClear(void);
```

psErrorGetStackSize shall return the number of errors on the stack. The entire error stack may be printed to an open file descriptor by calling psErrorStackPrint (or psErrorStackPrintV); if and only if there are current errors, the printf-style string format is first printed to the file descriptor fd. In this printout, error codes shall be replaced by their string equivalents as defined in the next section. Note that these are also available in the psErr structure. The successive lines of the traceback should be indented by an additional space. psErrorStackPrintV must not invoke va_end.

```
void psErrorStackPrint(FILE *fd, const char *format, ...);
void psErrorStackPrintV(FILE *fd, const char *format, va_list va);
```

Any error codes less then or equal to PS_ERR_BASE (see next section) must be taken to be valid values of errno, and psErrorStackPrint must print the value returned by strerror if such error codes are encountered.

psErrorCodeLast returns the last error code:

```
psErrorCode psErrorCodeLast(void)
```

The routine psErrorCodeString returns the string associated with an error code:

```
const char *psErrorCodeString(psErrorCode code);
```

2.5.1 Error Codes

Both error codes for PSLib and error codes for projects that use PSLib may be registered. In the former case, the error codes must be registered on initialisation (see `psLibInit`), whereas in the latter case, they must be explicitly registered by the programmer.

2.5.1.1 Registering error codes

PSLib and any project needed to use PSLib must define the necessary error codes and associated message strings. An array of error codes may be registered with the PSLib error handler using the function:

```
void psErrorRegister(const psErrorDescription *errors, int errorCode);
```

where the errors are represented internally as follows:

```
typedef struct {
    psErrorCode code;                   ///< An error code
    const char *description;            ///< the associated description
} psErrorDescription;
```

PSLib internal errors must be registered with the function `psErrorRegister`, which should be called as part of the program initialization to set up the error codes. It is left to the external project to produce their own error registration functions which must also be called during initialization. There is a clear need to coordinate the choice of error numbers. It is expected that error code ranges for different projects must be managed by the Project Office within Pan-STARRS.

2.5.1.2 Error Codes for PSLib

For ease of maintenance, error codes for PSLib must be defined by an auxiliary file, conventionally named *psErrorCodes.dat*. The format of this file must consist of a number of lines, each of the form:

```
NAME [ = value ][,] STRING
```

where `[ = value]` and the comma are optional, and no spaces are significant except in the STRING. Comments extend from # to the end of the line (except that a \# must be replaced by # and not taken to start a comment). For example,

```
#
# This file is used to generate psErrorClasses.h
#
NONE = 0,               not an error; must be 0
BASE = 256,             first value we use; should avoid errno conflicts
UNKNOWN,                unknown error
# This is a comment, and is ignored.
IO,                     I/O error
BADFREE,                bad argument to psFree()
MEMORY_CORRUPTION,      memory corruption detected
```

The values `NONE = 0` and UNKNOWN must be present.

A script, called from the Makefiles, must generate two files, *psErrorCodes.h* and *psErrorCodes.c* from the input file *psErrorCodes.dat*. *psErrorCodes.h* must define an enumerated type `psErrorCode` with elements `PS_ERR_NAME` and values as specified in *psErrorCodes.dat*, e.g.

```
#if !defined(PS_ERROR_CODES_H)
#define PS_ERROR_CODES_H

typedef enum {
    PS_ERR_NONE = 0,
    PS_ERR_BASE = 256,
    PS_ERR_UNKNOWN,
    PS_ERR_IO,
    PS_ERR_BADFREE,
    PS_ERR_MEMORY_CORRUPTION,
    PS_ERR_N_ERR_CLASSES,
} psErrorCode;
#endif
```

psErrorCodes.c must define the necessary functions to register the error codes.

This script will likely be of use to the user, and so it shall be installed as part of PSLib.

2.6 Abort

`psAbort`, must call `psLogMsgV` with a level of `PS_LOG_ABORT`, and then call `abort`.

```
void psAbort(const char *name, const char *format, ...);
```

2.7 Command-line arguments

The following functions are provided to aid parsing of command-line arguments:

```
int psArgumentGet(int argc, char **argv, const char *arg);
bool psArgumentRemove(int argnum, int *argc, char **argv);
```

`psArgumentGet` shall return the index of the element in the argument list, `argv` with number of entries `argc`, that matches the specified argument, `arg`, or zero if there is no match.

`psArgumentRemove` shall remove from the argument list (`argv` with number of entries `argc`) the argument whose index is `argnum`. The number of entries in the argument list shall be decremented. The function shall return `true` if the `argnum` is in the argument list, and `false` otherwise.

```
int psArgumentVerbosity(int *argc, char **argv);
```

`psArgumentVebosity` shall implement the various verbosity controls under the following guidelines:

- `-v` switch shall set the log level to 3.

- `-vv` switch shall set the log level to 4.
- `-vvv` switch shall set the log level to 5.
- `-logfmt someFormat` switch shall set the log format to `someFormat`.
- `-trace facility level` switch shall set the trace level for the specified `facility` to the specified `level`.
- `-trace-levels` switch shall print the trace levels as currently set.

The above arguments shall be removed from the argument list as they are processed. The function shall return the resultant logging level.

```
bool psArgumentParse(psMetadata *arguments, int *argc, char **argv);
```

`psArgumentParse` shall parse the command line arguments (supplied via `argc, argv`) into a metadata container of `arguments`. The input `arguments` shall contain the list of possible arguments as the keywords providing the default values (of the appropriate type). As matching arguments are found on the command line, the values shall be read into the `arguments` metadata, with the appropriate type.

An argument may be specified multiple times on the command line (e.g., `-arg 1 -arg 2`) if the argument is specified using a `PS_DATA_METADATA_MULTI` list (i.e., by specifying a type with the `PS_META_DUPLICATE_OK` flag on creation of the `arguments`). An argument may take more than one parameter (e.g., `-arg 1 2 3`) by adding a `psMetadata` to the `arguments`, containing an entry for each of the parameters. A boolean argument shall be set to `true` by the presence of the argument itself (no value is specified). The arguments and their values shall be removed from the list of command line arguments as they are processed. The function shall return `false` without modifying the `arguments` if any argument (i.e., a string beginning with a dash; but not a string that does not start with a dash, which is likely a mandatory argument for the program; the purpose of this requirement is so that the default values are preserved) was encountered that is not present in the `arguments`.

```
void psArgumentHelp(psMetadata *arguments);
```

`psArgumentHelp` shall print to `stdout` a guide to the command-line `arguments` (containing the same information as for `psArgumentParse`) of the form:

```
Optional arguments, with default values:
    -names     (FRED)     This is the comment from the first value
               (JIM)      This is the comment from the second value
               (BOB)      This is the comment from the third value
    -answer    (42)       This is a comment
    -truth     (FALSE)    This is another comment
```

Here the `arguments` contains three keywords, `names, magic, truth`. `names` has three values, `FRED, JIM, BOB`, stored as a `PS_DATA_METADATA_MULTI`, each with the comment as shown.

It will be the responsibility of the user to supply information for the mandatory arguments. An example follows:

```
int main(int argc, char *argv[])
{
    // Parse optional command-line arguments
```

```
    psMetadata *arguments = psMetadataAlloc(); // The arguments, with default values
    psMetadataAdd(arguments, PS_LIST_TAIL, "-string", PS_DATA_STR | PS_META_DUPLICATE_OK,
                  "Test strings", NULL); // Multiple strings are permitted
    psMetadataAdd(arguments, PS_LIST_TAIL, "-bool", PS_TYPE_BOOL, "Test bool", false);
    psMetadata *intParams = psMetadataAlloc(); // Parameters for -int
    psMetadataAdd(intParams, PS_LIST_TAIL, "int1", PS_TYPE_S32, "Test integer 1", 1);
    psMetadataAdd(intParams, PS_LIST_TAIL, "int2", PS_TYPE_S32, "Test integer 2", 2);
    psMetadataAdd(intParams, PS_LIST_TAIL, "int3", PS_TYPE_S32, "Test integer 3", 3);
    psMetadataAdd(arguments, PS_LIST_TAIL, "-int", PS_DATA_METADATA, "Test integers", intParams);
    psFree(intParams); // Drop reference
    psMetadataAdd(arguments, PS_LIST_TAIL, "-float", PS_TYPE_F32, "Test float", 1.234567);
    if (! psArgumentParse(arguments, &argc, argv) || argc != 3) {
printf("\nName of the program here\n\n");
printf("Usage: %s INPUT OUTPUT\n\n", argv[0]);
psArgumentHelp(arguments);
psFree(arguments);
exit(EXIT_FAILURE);
    }
    const char *inputName = argv[1]; // Name of input
    const char *outputName = argv[2]; // Name of output
    printf("Success: %s %s\n", inputName, outputName);

    psMetadataItem *strings = psMetadataLookup(arguments, "-string"); // This is a MULTI
    psListIterator *stringsIter = psListIteratorAlloc(strings->data.V, PS_LIST_HEAD, false);
    psString string;
    while ((string = psListGetAndIncrement(stringsIter))) {
        printf("String: %s\n", string);
    }
    psFree(stringsIter);
    float floating = psMetadataLookupF32(NULL, arguments, "-float");
    bool boolean = psMetadataLookupBool(NULL, arguments, "-bool");
    printf("Float: %f\n", floating);
    printf("Boolean: %d\n", boolean);

    psMetadata *ints = psMetadataLookupMD(NULL, arguments, "-int");
    int int1 = psMetadataLookupS32(NULL, ints, "int1");
    int int2 = psMetadataLookupS32(NULL, ints, "int2");
    int int3 = psMetadataLookupS32(NULL, ints, "int3");
    printf("Integer 1: %d\n", int1);
    printf("Integer 2: %d\n", int2);
    printf("Integer 3: %d\n", int3);
}
```

The above program should produce, on being given no arguments:

```
Name of the program here

Usage: ./program INPUT OUTPUT

Optional arguments, with default values:
    -string    ()               Test strings
    -bool      (FALSE)          Test bool
    -int       (1)              Test integer 1
               (2)              Test integer 2
               (3)              Test integer 3
    -float     (1.234567e+00)   Test float
```

and called with: `./program -string foo -string bar -float 9.876543 -int 4 5 6 -bool input output`, then it should produce:

```
Success: input output
String: foo
String: bar
```

```
Float: 9.876543
Boolean: 1
Integer1: 4
Integer2: 5
Integer3: 6
```

3 Containers

We require general data containers, so that associated values (e.g. the elements of an array) may be connected as a whole. We require the following types of containers:

- Arrays;
- Doubly-linked lists;
- Hashes;
- Pixel lists;
- Bit sets;
- Metadata; and
- Lookup tables.

The reference counter for a pointer shall be incremented when it is placed in a container, and decremented when the pointer is it is removed.

3.1 Arrays

We require an order collection of unspecified data elements. We define `psArray` to carry such a collection:

```
typedef struct {
    long n;                             ///< size of array
    const long nalloc;                  ///< allocated data block
    psPtr *data;                        ///< pointer to data block
    void *lock;                         ///< Optional lock for thread safety
} psArray;
```

In this structure, the argument `n` is the length of the array (the number of elements); `nalloc` is the number of elements allocated ($nalloc \geq n$). The allocated memory is pointed to by `data`. The structure is associated with a constructor and a destructor:

```
psArray *psArrayAlloc(long nalloc);
psArray *psArrayRealloc(psArray *array, long nalloc);
```

In these functions, `nalloc` is the number of elements to allocate. In `psArrayAlloc`, the value of `psArray.n` is initially set to zero. Users may choose to restrict the data range after the `psArrayAlloc` function is called. For `psArrayRealloc`, if the value of `nalloc` is smaller than the current value of `psArray.n`, then `psArray.n` is set to `nalloc`, the array is adjusted down to match `nalloc`, and the extra elements are dropped and freed if necesitated by the reference counter. If `nalloc` is larger than the current value of `psArray.n`, `psArray.n` is left intact. If the value of `array` is `NULL`, then `psArrayRealloc` must return an error.

Since `psArray` stores pointers, values on the array shall always be initialised to `NULL` on `psArrayAlloc`. `psArrayRealloc` shall initialise values to `NULL` where the array has been grown.

```
void psArrayElementsFree(psArray* array);
```

`psArrayElementsFree` shall free all elements on the `array`.

```
psArray *psArrayAdd(psArray *array, long delta, psPtr data);
```

This function adds a value to the end of an array. If the current length of the array (`psArray.n`) is at the limit of the allocated space, additional space is allocated. The value of `delta` defines how many elements to add on each pass (if this value is less than 1, 10 shall be used).

```
bool psArrayRemove(psArray *array, const psPtr data);
```

This function removes all entries of `value` in the `array`, reducing the total number of elements of `array` as needed. Returns `TRUE` if any elements were removed, otherwise `FALSE`.

```
long psArrayLength(const psArray *array);
```

This function returns the length of the array (`psArray.n`).

```
bool psArraySet(psArray *array, long position, psPtr data);
psPtr psArrayGet(const psArray *array, long position);
```

These accessor functions are provided as a convenience to the user. `psArraySet` sets the value of the `in` array at the specified `position` to `value`, returning `true` if successful. `psArrayGet` returns the value of the `in` array at the specified `position`. A negative `position` means index from the end.

```
typedef int (*psComparePtrFunc) (
    const void **a,                    ///< first comparison target
    const void **b                     ///< second comparison target
);

psArray *psArraySort(psArray *array, psComparePtrFunc func);
```

An array may be sorted using `psArraySort`, which requires the specification of a comparison function to specify how the objects on the list should be sorted. The motivation is primarily to be able to iterate on a sorted list of keys from a hash. The `array` is sorted in-place.

3.1.1 Comparison functions

We specify several comparison functions as a convenience to the user. The `psCompareTypePtr` functions are intended for sorting where the container has a pointer to the value (e.g., `psList`). The functions return an integer less than, equal to, or greater than zero if `a` is is less than, equal to, or greater than `b`, respectively (`qsort` man page).

```
int psCompareS8Ptr(const void **a, const void **b);
int psCompareS16Ptr(const void **a, const void **b);
int psCompareS32Ptr(const void **a, const void **b);
int psCompareS64Ptr(const void **a, const void **b);
int psCompareU8Ptr(const void **a, const void **b);
int psCompareU16Ptr(const void **a, const void **b);
int psCompareU32Ptr(const void **a, const void **b);
int psCompareU64Ptr(const void **a, const void **b);
int psCompareF32Ptr(const void **a, const void **b);
int psCompareF64Ptr(const void **a, const void **b);
int psCompareDescendingS8Ptr(const void **a, const void **b);
int psCompareDescendingS16Ptr(const void **a, const void **b);
int psCompareDescendingS32Ptr(const void **a, const void **b);
int psCompareDescendingS64Ptr(const void **a, const void **b);
int psCompareDescendingU8Ptr(const void **a, const void **b);
int psCompareDescendingU16Ptr(const void **a, const void **b);
int psCompareDescendingU32Ptr(const void **a, const void **b);
int psCompareDescendingU64Ptr(const void **a, const void **b);
int psCompareDescendingF32Ptr(const void **a, const void **b);
int psCompareDescendingF64Ptr(const void **a, const void **b);
int psCompareS8(const void *a, const void *b);
int psCompareS16(const void *a, const void *b);
int psCompareS32(const void *a, const void *b);
int psCompareS64(const void *a, const void *b);
int psCompareU8(const void *a, const void *b);
int psCompareU16(const void *a, const void *b);
int psCompareU32(const void *a, const void *b);
int psCompareU64(const void *a, const void *b);
int psCompareF32(const void *a, const void *b);
int psCompareF64(const void *a, const void *b);
int psCompareDescendingS8(const void *a, const void *b);
int psCompareDescendingS16(const void *a, const void *b);
int psCompareDescendingS32(const void *a, const void *b);
int psCompareDescendingS64(const void *a, const void *b);
int psCompareDescendingU8(const void *a, const void *b);
int psCompareDescendingU16(const void *a, const void *b);
int psCompareDescendingU32(const void *a, const void *b);
int psCompareDescendingU64(const void *a, const void *b);
int psCompareDescendingF32(const void *a, const void *b);
int psCompareDescendingF64(const void *a, const void *b);
```

3.2 Doubly-linked lists

Pan-STARRS shall support doubly linked lists through a type `psList`:

```
typedef struct {
   long n;                          ///< number of elements on list
   psListElem *head;                ///< first element on list (may be NULL)
   psListElem *tail;                ///< last element on list (may be NULL)
   psArray *iterators;              ///< array of psListIterator: iteration cursors
   void *lock;                      ///< Optional lock for thread safety
} psList;
```

The type `psList` represents the container of the list. It has a pointer to the first element in the linked list (`head`), a pointer to the last element in the list (`tail`), an array of iteration cursors, (`iterators`), and an entry to define the number of elements in the list (`n`).

The elements of the list are defined by the type `psListElem`:

```
typedef struct psListElem {
   struct psListElem *prev;            ///< previous link in list
   struct psListElem *next;            ///< next link in list
   psPtr data;                         ///< real data item
} psListElem;
```

which includes a pointer to the next element in the list (`next`), the previous element in the list (`prev`), and a `void` pointer to whatever data is represented by this list element. The following supporting functions are required:

```
psList *psListAlloc(psPtr data);
```

Create a list. This function may take a pointer to a data item, or it may take `NULL`. The allocator creates both the `psList` and the first element in the list, pointed to by both `psList.head` and `psList.tail`. If the data entry is `NULL`, then an empty list, with both pointers set to `NULL` should be created.

The destructor function for `psList` must call `psFree` for all the the data associated with the list.

```
long psListLength(const psList *list);
```

Return the length of the list (`psList.n`).

All data items placed onto lists must have their reference counters (section 2.3.7) incremented. When elements are removed from a list, they must have their reference counters decremented. The action of retrieving data from a list (with one of the three `psListGet` functions) is considered "borrowing" the reference, so no action is performed on the reference counter.

Iteration on the list shall be achieved by means of a list iterator type:

```
typedef struct {
    psList *list;                      ///< List iterator works on
    psListElem *cursor;                ///< The current iterator cursor
    bool offEnd;                       ///< Is the iterator off the end?
    long index;                        ///< Index of iterator, to assist performance
    bool mutable;                      ///< Is it permissible to modify the list?
} psListIterator;
```

The `psListIterator` keeps track of which list element the iterator `cursor` is currently pointing at. `index` is the index of the list iterator, which is used to assist performance when using numerical locations. The boolean member, `offEnd`, indicates whether the iterator has progressed off the end of the list (i.e., beyond the last item). The boolean `mutable` specifies whether it is permissible to modify the list pointed to by the iterator. `psListAddBefore` and `psListAddAfter` are not permitted to modify a list that is not `mutable` (i.e., only the `psListGetAndIncrement` and `psListGetAndDecrement` operations are permissible for a non-`mutable` list).

The corresponding constructor shall be:

```
psListIterator *psListIteratorAlloc(psList *list, long location, bool mutable);
```

Here, `list` is the `psList` on which the iterator will iterate, and `location` is the initial starting point, and may be a numerical index or it may be one of the special values: `PS_LIST_HEAD` or `PS_LIST_TAIL`, which are defined as 0 and

-1, respectively; a negative index is interpreted as relative to the end of the list. The boolean `mutable` specifies whether it is permissible to modify the list pointed to by the iterator.

The destructor for `psListIterator` shall, after freeing the `psListIterator`, also reorganise the `iter` array (replacing the element being removed with the last element) and resizing the array appropriately.

A list `iterator` shall be set to a specific `location` on the list upon calling `psListIteratorSet`:

```
bool psListIteratorSet(psListIterator *iterator, long location);
```

Again, the `location` may be a numerical index or it may be one of the special values: `PS_LIST_HEAD` or `PS_LIST_TAIL`, which are defined as 0 and -1, respectively; a negative index is interpreted as relative to the end of the list. The function shall return `true` if the reset was successful, or `false` otherwise.

```
bool psListAdd(psList *list, long location, psPtr data);
bool psListAddAfter(psListIterator *iterator, psPtr data);
bool psListAddBefore(psListIterator *iterator, psPtr data);
```

the first function, `psListAdd`, adds an entry to the `list` and returns a boolean giving the success or failure of the operation. The value of `location` may be a numerical index the `data` is to inhabit (if `location` is greater than the number of items on the list, then the function shall generate a warning and add the `data` to the tail) or it may be one of the special values: `PS_LIST_HEAD` or `PS_LIST_TAIL`, which are defined as 0 and -1, respectively; a negative index is interpreted as relative to the end of the list. The other two functions, `psListAddAfter` and `psListAddBefore` specify that the `data` shall be added after or before (respectively) the current cursor position of the `iterator`.

```
psPtr psListGet(psList *list, long location);
psPtr psListGetAndIncrement(psListIterator *iterator);
psPtr psListGetAndDecrement(psListIterator *iterator);
```

A data item may be retrieved from the list with these functions. The first function, `psListGet` simply returns the value specified by its `location`, which may be a numerical index or it may be one of the special values: `PS_LIST_HEAD = 0` or `PS_LIST_TAIL = -1`; negative indices are interpreted as relative to the end of the list. The other two functions, `psListGetAndIncrement` and `psListGetAndDecrement` return the item under the iteration cursor before advancing to the next or previous item, respectively.

In the event that the iteration cursor is at the tail of the list, `psListGetAndIncrement` shall return the tail item and then set the `cursor` to `NULL` and `offEnd` to `true`. In the event that the iteration cursor is at the head of the list, `psListGetAndDecrement` shall return the head item and then set the `cursor` to `NULL` (and `offEnd` should already be `false`). In the event that the iteration `cursor` is `NULL`, `psListGetAndIncrement` and `psListGetAndDecrement` shall return `NULL`, and advance the iteration `cursor` only if the intended direction places the cursor back on the list; otherwise a warning shall be generated, and no change shall be made. If `psListGetAndDecrement` was called with `offEnd` as `true`, then `offEnd` shall also be toggled back to `false` to indicate that the `cursor` is no longer off the end of the list.

```
bool psListRemove(psList *list, long location)
bool psListRemoveData(psList *list, psPtr data);
```

A data item may be removed from the list with these functions. For `psListRemove`, the value of `location` may be the numerical index or it may be one of the special values: `PS_LIST_HEAD` or `PS_LIST_TAIL`, which are defined as 0

and -1, respectively; a negative index is interpreted as relative to the end of the list. For `psListRemoveData`, the data item to be removed is identified by matching the pointer value with `psPtr data`. The functions return a value of `true` if the operation was successful, and `false` otherwise. In both cases, if any iterators are currently pointing at the item to be removed, the item shall be removed and those iterators pointing at it shall be moved to the next, and the function shall return `true`. If the item to be removed is not on the list, an error shall be generated and the function shall return `false`.

```
psArray *psListToArray(const psList *list);
psList  *psArrayToList(const psArray *array);
```

These two functions are available to convert between the `psList` and `psArray` containers. These functions do not free the elements or destroy the input collection. Rather, they increment the reference counter for each of the elements.

```
psList *psListSort(psList *list, psComparePtrFunc func);
```

A list may be sorted using `psListSort`, which requires the specification of a comparison function to specify how the objects on the list should be sorted. The motivation is primarily to be able to iterate on a sorted list of keys from a hash. The `list` is sorted in-place.

3.3 Hash Tables

Hash tables are critical for quick retrieval of text-based data. The concept is as follows: Given a large collection of text strings, it is inefficient to search for a particular entry by performing a basic string comparison on all entries until a match is found. Even if a single list is sorted, we will still spend a substantial amount of time iterating across the entries in the list. In a hash table, we define an operation, the hash function, which uses the bytes of the string to construct a numerical value, the hash value. The hash value is defined to have a limited range of N values. The hash table consists of N buckets, each of which contains a list of the strings whose hash value corresponds to the bucket number. Searching for a specific string involves calculating the hash value for the string, going to the appropriate bucket, and searching through the corresponding list until the string is matched.

For PSLib, we define a hash table and hash buckets as follows: [1]

```
typedef struct {
    long n;                             ///< number of buckets
    psHashBucket **buckets;             ///< the buckets themselves
    void *lock;                         ///< Optional lock for thread safety
} psHash;
```

where `n` is the number of buckets defined for the hash functions, and `buckets` is an array of pointers to the individual buckets, each of which is defined by:

```
typedef struct psHashBucket {
    char *key;                          ///< key for this item of data
    psPtr data;                         ///< the data itself
    struct psHashBucket *next;          ///< list of other possible keys
} psHashBucket;
```

[1] We choose not to use the POSIX function `hcreate`, `hdestroy`, and `hsearch` as they only support a single hash table at any one time.

where each bucket contains the value of the `key`, a pointer to the `data`, and a pointer to the `next` list entry in the bucket (in the event that two or more keys have the same hash value).

A hash table is created with the following function:

```
psHash *psHashAlloc(long nalloc);
```

which allocates the space for the hash table, creating and initializing `n` hash buckets.

The destructor for `psHash` must free all data associated with a complete hash table.

A data item may be added to the hash table with the function:

```
bool psHashAdd(psHash *hash, const char *key, psPtr data);
```

In this function, the value of the string `key` is used to construct the hash value, find the appropriate bucket set, and add the new element `data` to the list. An existing element with the same value of `key` is destroyed using its registered destructor (`psMemBlock`). The return value of the function is a boolean defining the success or failure of the operation.

The data associated with a given key may be found with the function:

```
psPtr psHashLookup(const psHash *hash, const char *key);
```

which returns the data value pointed to by the element associated with `key`, or the value `NULL` if no match is found. Similarly, a specific key may be removed (deleted) with the function:

```
bool psHashRemove(psHash *hash, const char *key);
```

The function returns a value of `true` if the operation was successfull, and `false` otherwise.

The function

```
psList *psHashKeyList(const psHash *hash);
```

returns the complete list of defined keys associated with the `psHash` table as a linked list.

```
psArray *psHashToArray(const psHash *hash);
```

This function places the data in a `psHash` into a `psArray` container. This function does not free the elements or destroy the input collection. Rather, it increments the reference counter for each of the elements. The resulting array does not have any information about the has key values, and the order is not significant.

3.4 Pixel Lists

Usually an image mask is the best way to carry information about what pixels mean what. However, in the case where the number of pixels in which we are interested is limited, it is more efficient to simply carry a list of pixels. An example of this is in the image combination code, where we want to perform an operation on a relatively small fraction of pixels, and it is inefficient to go through an entire mask image checking each pixel.

```
typedef struct {
    float x;                    // x coordinate
    float y;                    // y coordinate
} psPixelCoord;

typedef struct {
    long n;                     // Number in use
    const long nalloc;          // Number allocated
    psPixelCoord *data;         // The pixel coordinates
    void *lock;                 // Lock for thread safety
} psPixels;

psPixels *psPixelsAlloc(long nalloc);
psPixels *psPixelsRealloc(psPixels *pixels, long nalloc);
```

`psPixelsAlloc` and `psPixelsRealloc` provide dynamic allocation and reallocation in a manner analogous to those provided by `psVectorAlloc` and `psVectorRealloc`.

```
long psPixelsLength(const psPixels *pixels);
```

`psPixelsLength` shall return the length of the pixel array (`psPixels.n`).

```
psPixels *psPixelsCopy(psPixels *out, const psPixels *pixels);
psPixels *psPixelsConcatenate(psPixels *out, const psPixels *pixels);
```

`psPixelsCopy` shall copy the contents of `pixels` to the `out`. In the event that `out` is `NULL`, a new `psPixels` shall be allocated, and the contents of `pixels` simply copied in. If `pixels` is `NULL`, the function shall generate an error and return `NULL`.

`psPixelsConcatenate` shall concatenate the `pixels` onto `out`. In the event that `out` is `NULL`, the function performs a `psPixelsCopy`, returning the copy. If `pixels` is `NULL`, the function shall generate an error and return `NULL`. The function shall take care to ensure that there are no duplicate pixels in `out` (since the order in which the pixels are stored is not important, the values may be sorted, allowing the use of a faster algorithm than a linear scan).

```
bool psPixelsSet(psPixels *pixels, long position, psPixelCoord value);
psPixelCoord psPixelsGet(const psPixels *pixels, long position);
```

These accessor functions are provided as a convenience to the user. `psPixelsSet` sets the value of the `pixels` array at the specified `position` to `value` (a `psPixelCoord` passed by value), returning `true` if successful. `psPixelsGet` returns the value of the `pixels` array at the specified `position`. A negative `position` means index from the end. In the event of an error, `psPixelsGet` shall return a `psPixelCoord` with components set to `NaN`.

3.5 BitSets

BitSets are required in order to turn options on and off. We require the capability to have a bitset of arbitrary length (i.e., not limited by the length of a `long`, say). The `psBitSet` structure is defined below. Note that the entry `bits` is an array of type `char` storing the bits as bits of each byte in the array, with 8 bits available for each byte in the array. Also note that the constructor is passed the number of required bits, which implies that `ceil(n/8)` bytes must be allocated. The bitset structure is define by:

```
typedef struct {
    long n;                             ///< Number of chars that form the bitset
    psU8 *bits;                         ///< The bits
    void *lock;                         ///< Optional lock for thread safety
} psBitSet;
```

We also require the corresponding constructor and destructor:

```
psBitSet *psBitSetAlloc(long nalloc);
```

where `n` is the requested number of bits.

Several basic operations on bitsets are required:

- Set a bit;
- Check if a bit is set; and
- `OR`, `AND` and `XOR` two bitsets.
- `NOT` a bitset.

The corresponding APIs are defined below:

```
psBitSet *psBitSetSet(psBitSet *bitSet, long bit);
psBitSet* psBitSetClear(psBitSet *bitSet, long bit);
psBitSet *psBitSetOp(psBitSet *outBitSet, const psBitSet *inBitSet1, const char *operator, const psBitSet *
psBitSet *psBitSetNot(psBitSet *outBitSet, const psBitSet *inBitSet);
bool psBitSetTest(const psBitSet *bitSet, long bit);
psString psBitSetToString(const psBitSet* bitSet);
```

`psBitSetSet` sets the specified `bit` in the `psBitSet`, and returns the updated bitset. The input bitset will be modified.

`psBitSetClear` clears the specified `bit` in the `bitSet` and returns the updated bitset. The input bitset will be modified.

`psBitSetOp` returns the `psBitSet` that is the result of performing the specified `operator` (one of `"AND"`, `"OR"`, or `"XOR"`) on `inBitSet1` and `inBitSet2`. If the output bitset `outBitSet` is `NULL`, it is created by the function.

`psBitSetNot` applies a unary `NOT` to a bitset, placing the answer in the bitset `out`, or creating a new bitset if `out` is `NULL`.

`psBitSetTest` returns a true value if the specified `bit` is set; otherwise, it returns a false value.

Finally, `psBitSetToString` returns a string representation of the specified `bits`.

3.6 Metadata

3.6.1 Conceptual Overview

Within PSLib, we provide a data structure to carry metadata and mechanisms to manipulate the metadata. Metadata is a general concept that requires some discussion. In any data analysis task, the ensemble of all possible data may be divided

into two or three classes: there is the specific data of interest, there is data which is related or critical but not the primary data of interest, and there is all of the other data which may or may not be interesting. For example, consider a simple 2D image obtained of a galaxy from a CCD camera on a telescope. If you want to study the galaxy, the specific data of interest is the collection of pixels. There are a variety of other pieces of data which are closely related and crucial to understanding the data in those pixels, such as the dimensions of the image, the coordinate system, the time of the image, the exposure time, and so forth. Other data may be known which may be less critical to understanding the image, but which may be interesting or desired at a later date. For example, the observer who took the image, the filter manufacturer, the humidity at the telescope, etc.

Formally, all of the related data which describe the principal data of interest are metadata. Note that which piece is the metadata and which is the data may depend on the context. If you are examining the pixels in an image, the coordinate and flux of an object may be part of the metadata. However, if you are analyzing a collection of objects extracted from an image, you may consider then pixel data simply part of the metadata associated with the list of objects.

There are various ways to handle metadata vs data within a programming environment. In C, it is convenient to use structures to group associated data together. One possibility is to define the metadata as part of the associated data structure. For example, the image data structure would have elements for all possible associated measurement. This approach is both cumbersome (because of the large number metadata types), impractical (because the full range of necessary metadata is difficult to know in advance) and inflexible (because any change in the collection of metadata requires addition of new structure elements and recompilation).

An alternative is to place the metadata in a generic container and use lookup mechanisms to extract the appropriate metadata when needed. An example of this is would be a text-based FITS header for an image read into a flat text buffer. In this implementation, metadata lookup functions could return the current value of, for example, NAXIS1 (the number of columns of the image) by scanning through the header buffer. This method has the benefits of flexibility and simplicity of programming interface, but it has the disadvantage that all metadata is accessed though this lookup mechanism. This may make the code less readable and it may slow down the access.

PSLib implements an intermediate solution to this problem. We specify a flexible, generic metadata container and access methods. Data types which require association with a general collection of metadata should include an entry of this metadata type. However, a subset of metadata concepts which are basic and frequently required may be placed in the coded structure elements. This approach allows the code to refer to the basic metadata concepts as part of the data structure (ie, `image.nx`), but also allows us to provide access to any arbitrary metadata which may be generated. As a practical matter, the choice of which entries are only in the metadata and which are part of the explicit structure elements is rather subjective. Any data elements which are frequently used should be put in the structure; those which are only infrequently needed should be left in the generic metadata.

There are some points of caution which must be noted. Any manipulation of the data should be reflected in the metadata where appropriate. This is always an issue of concern. For example, consider an image of dimensions `nx, ny`. If a function extracts a subraster, it must change the values of `nx, ny` to match the new dimensions. What should it do to the corresponding metadata? Clearly, it should change the corresponding value which defines `nX, nY`. However, it is not quite so simple: there may be other metadata values which depend on those values. These must also be changed appropriately. What if the metadata element points to a copy of the metadata which may be shared by other representations of the image? These must be treated differently because the change would invalidate those other references. Care must be taken, therefore, when writing functions which operate on the data to consider all of the relevant metadata entries which must also be updated.

A related issue is the definition of metadata names. Entries in a structure have the advantage of being hardwired: every instance of that structure will have the same name for the same entry. This is not necessarily the case with a more flexible metadata container. The image exposure time is a notorious example in astronomy. Different observatories use different

header keywords (ie, metadata names) for the same concept of the exposure time (`EXPTIME`, `EXPOSURE`, `OPENTIME`, `INTTIME`, etc). Any system which operates on these metadata needs to address the issue of identifying these names. This issue seems like an argument for hardwiring metadata in the structure, but in fact it does not present such a strong case. If the metadata are hardwired, some function will still have to know how to interpret the various names to populate the structure. The concept can still be localized with generic metadata containers by including abstract metadata names within the code which are tied to the various implementations-specific metadata names.

3.6.2 Metadata Representation

This section addresses the question of how Pan-STARRS metadata should be represented in memory, not how it should be represented on disk.

We define an item of metadata with the following structure:

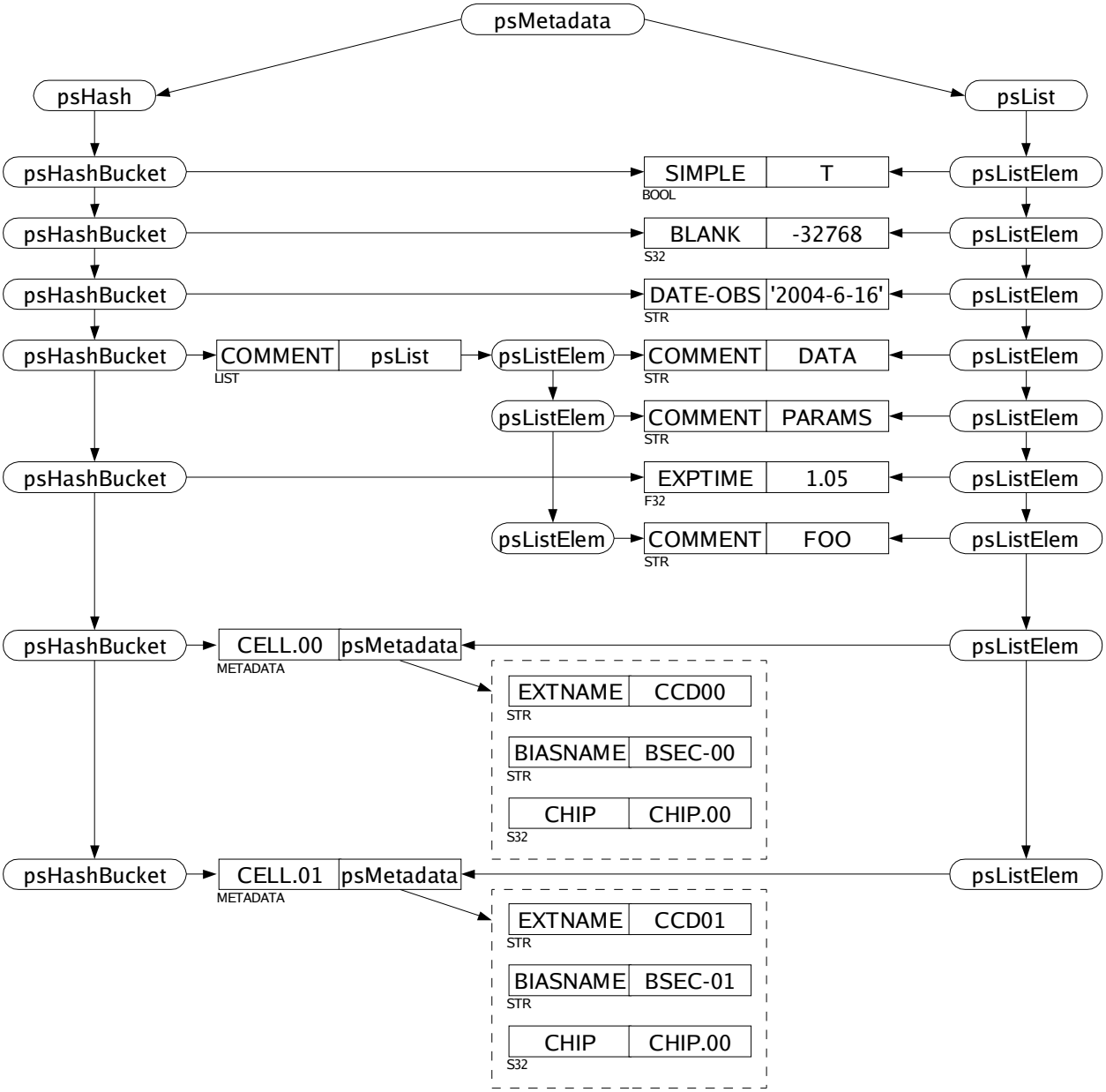

Figure 1: Metadata Structures

```
typedef struct {
    const psS32 id;                     ///< unique ID for this item
    psString name;                      ///< Name of item
    psDataType type;                    ///< type of this item
    union {
        psBool B;                       ///< boolean value
        psS8  S8;                       ///< integer data
        psS16 S16;                      ///< integer data
        psS32 S32;                      ///< integer data
        psS64 S64;                      ///< integer data
        psU8  U8;                       ///< integer data
        psU16 U16;                      ///< integer data
        psU32 U32;                      ///< integer data
        psU64 U64;                      ///< integer data
        psF32 F32;                      ///< floating-point data
        psF64 F64;                      ///< double-precision data
        psList *list;                   ///< psList entry
        psMetadata *md;                 ///< psMetadata entry
        psPtr V;                        ///< other type
    } data;                             ///< value of metadata
    psString comment;                   ///< optional comment ("", not NULL)
} psMetadataItem;
```

The `id` is a unique identifier for this item of metadata; experience shows that such tags are useful. The entry `name` specifies the name of the metadata item. The value of the metadata is given by the union `data`, and may be of type `psU8`, `psU16`, `psU32`, `psS8`, `psS16`, `psS32`, `psF32`, `psF64`, or an arbitrary rich structure pointed at by the `void` pointer `V`. A character string comment associated with this metadata item may be stored in the element `comment`. The `type` entry specifies how to interpret the type of the data being represented, given by the enumerated type `psDataType`.

Note that the `name` and `comment` must be allocated by the constructor using, e.g., `psStringCopy`.

A collection of metadata is represented by the `psMetadata` structure:

```
typedef struct {
    psList *list;                       ///< list of psMetadataItem
    psHash *hash;                       ///< hash table of the same metadata
    void *lock;                         ///< Optional lock for thread safety
} psMetadata;
```

The type `psMetadata` is a container class for metadata. Note that there are in fact *two* representations of the metadata (each `psMetadataItem` appears on both). The first representation employs a doubly-linked list that allows the order of the metadata to be preserved (e.g., if FITS headers are read in a particular order, they should be written in the same order). The second representation employs a hash table which allows fast look-up given a specific metadata keyword.

Certain metadata names (such as the FITS keywords `COMMENT` and `HISTORY` in a FITS header) may be repeated with different values. In such a case, the `psMetadata.list` structure contains the entries in their original sequence with duplicate keys. The `psMetadata.hash` entries, which are required to have unique keys, would have a single entry with the keyword of the repeated key, with the value of `psDataType` set to `PS_DATA_METADATA_MULTI`, and the `psMetadataItem.data` element pointing to a `psList` containing the actual entries. If `psMetadataItemAlloc` is called with the type set to `PS_DATA_METADATA_MULTI`, such a repeated key is created. In this case, the data value passed to `psMetadataItemAlloc` (the quantity in ellipsis) must be `NULL`. An empty `psMetadataItem` with the given keyword is created to hold future entries of that keyword.

As a convenience to the user, the following type-specific functions are also defined:

```
psMetadataItem* psMetadataItemAllocStr(const char* name, const char* comment, const char* value);
psMetadataItem* psMetadataItemAllocF32(const char* name, const char* comment, psF32 value);
psMetadataItem* psMetadataItemAllocF64(const char* name, const char* comment, psF64 value);
psMetadataItem* psMetadataItemAllocS32(const char* name, const char* comment, psS32 value);
psMetadataItem* psMetadataItemAllocBool(const char* name, const char* comment, bool value);
psMetadataItem* psMetadataItemAllocPtr(const char* name, psDataType type, const char* comment, psPtr value)
```

3.6.3 Metadata APIs

```
psMetadata *psMetadataAlloc(void);
```

The constructor for the collection of metadata, `psMetadata`, simply returns an empty metadata container (employing the constructors for the doubly-linked list and hash table). The destructor needs to free each of the `psMetadataItem`s.

```
psMetadataItem *psMetadataItemAlloc(const char *name, psDataType type, const char *comment, ...);
psMetadataItem *psMetadataItemAllocV(const char *name, psDataType type, const char *comment, va_list list);
```

The allocator for `psMetadataItem` returns a full `psMetadataItem` ready for insertion into the `psMetadata`. The `name` entry specifies the name to use for this metadata item, and may include `sprintf`-type formating codes. The `comment` entry is a fixed string which is used for the comment associated with this metadata item. The metadata data and the arguments to the `name` formatting codes are passed, in that order (metadata pointer first), to `psMetadataItemAlloc` as arguments following the comment string. The data must be a pointer for any data types which are stored in the element `data.void`, while other data types are passed as numeric values. All `data.void` types may be set to have a value of `NULL`. The argument list must be interpreted appropriately by the `va_list` operators in the function.

```
bool psMetadataAddItem(psMetadata *md, const psMetadataItem *item, int location, psS32 flags);
bool psMetadataAdd(psMetadata *md, long location, const char *name, int format, const char *comment, ...);
bool psMetadataAddV(psMetadata *md, long location, const char *name, int format, const char *comment,
                    va_list list);
```

Items may be added to the metadata in one of two ways — firstly, an item may be added by appending a `psMetadataItem` which has already been created; and secondly by directly providing the data to be appended. In both cases, the return value defines the success (`true`) or failure of the operation. The second function, `psMetadataAdd` takes a pointer or value which is interpreted by the function using variadic argument interpretation. The third version is the `va_list` version of the second function. All three functions take a parameter, `location`, which specifies where in the list to place the element, following the conventions for the `psList`. The entry `mode` for `psMetadataAddItem` is a bit mask constructed by OR-ing the allowed option flags (eg, `PS_DATA_REPLACE`) which specify minor variations on the behavior. The `format` entry, which specifies both the metadata type and the optional flags, is constructed by bit-wise OR-ing the appropriate `psDataType` and allowed option flags. Care should be taken not to leak memory when appending an item for which the key already exists in the metadata (and is not `PS_DATA_METADATA_MULTI`).

```
typedef enum {                          ///< option flags for psMetadata functions
    PS_META_DEFAULT         = 0,        ///< default behavior (0x0000) for use in mode above
    PS_META_REPLACE         = 0x1000000, ///< allow entry to be replaced
    PS_META_NO_REPLACE      = 0x2000000, ///< duplicate entry is silently skipped
    PS_META_DUPLICATE_OK    = 0x4000000, ///< allow duplicate entries
    PS_META_NULL            = 0x8000000  ///< psMetadataItem.data is a NULL value
} psMetadataFlags;
```

The functions above take option flags which modify the behavior when metadata items are added to the metadata list. These flags must be bit-exclusive of those used above for the psDataTypes. The flags have the following meanings:

PS_META_DEFAULT: This is the zero bit mask, to allow the default behavior for psMetadataAddItem above. If this is OR-ed with a psDataType, the result is as if no OR-ing took place.

PS_META_REPLACE: Replace an existing, unique entry. If the given metadata item exists in the metadata collection, and is not of type PS_META_MULTI, then the item replaces the existing entry.

PS_META_DUPLICATE_OK: Allow the new metadata item key to be a duplicate (ie, PS_DATA_METADATA_MULTI). If an existing item with the same key is already PS_DATA_METADATA_MULTI, the new item is added to the PS_DATA_METADATA_MULTI list. If the existing item is not PS_DATA_METADATA_MULTI, a PS_DATA_METADATA_MULTI list is created to contain both the existing item and the new item. The original entry's location on the psMetadata.list must be maintained.

PS_META_NULL: Indicates that psMetadataItem.data should be ignored and that the the current value is "NULL" or undefined. The psMetadataItem must have a proper type set and it's data field shall have a valid value. e.g. A type of PS_DATA_STR would require that its data is set to NULL.

There are several of cases to handle for duplication of an existing key by a new key, some identified above. The following situations must also be handled:

If the new key already exists, but is not PS_DATA_METADATA_MULTI, and the new item is not flagged as either PS_META_DUPLICATE_OK or PS_META_REPLACE, an error is raised.

If the new key already exists, and the existing item is PS_DATA_METADATA_MULTI, the new item is added to the MULTI list. Note that if the new item is also of type PS_DATA_METADATA_MULTI, no action is taken, but a successful exit status is returned (the action of adding a PS_DATA_METADATA_MULTI item to the metadata is equivalent to setting that key to be tagged as PS_DATA_METADATA_MULTI. If it is *already* PS_DATA_METADATA_MULTI, this effect has already been achieved).

An example of code to use these metadata APIs to generate the structure seen in Figure 1 is given below.

```
md = psMetadataAlloc();

psMetadataAdd(md, PS_LIST_TAIL, "SIMPLE",   PS_DATA_BOOL, "basic fits",            TRUE);
psMetadataAdd(md, PS_LIST_TAIL, "BLANK",    PS_DATA_S32,  "invalid pixel data",    -32768);
psMetadataAdd(md, PS_LIST_TAIL, "DATE-OBS", PS_DATA_STR,  "observing date UT", "   2004-6-16");
psMetadataAdd(md, PS_LIST_TAIL, "COMMENT",  PS_DATA_LIST, "head of comment block", NULL);
psMetadataAdd(md, PS_LIST_TAIL, "COMMENT",  PS_DATA_STR,  "",                      "DATA");
psMetadataAdd(md, PS_LIST_TAIL, "COMMENT",  PS_DATA_STR,  "",                      "PARAMS");
psMetadataAdd(md, PS_LIST_TAIL, "EXPTIME",  PS_DATA_F32,  "exposure time (sec)",   1.05);
psMetadataAdd(md, PS_LIST_TAIL, "COMMENT",  PS_DATA_STR,  "",                      "FOO");

cell = psMetadataAlloc();
psMetadataAdd(cell, PS_LIST_TAIL, "EXTNAME",  PS_DATA_STR,  "",                    "CCD00");
psMetadataAdd(cell, PS_LIST_TAIL, "BIASNAME", PS_DATA_STR,  "",                    "BSEC-00");
psMetadataAdd(cell, PS_LIST_TAIL, "CHIP",     PS_DATA_STR,  "",                    "CHIP.00");
psMetadataAdd(md,   PS_LIST_TAIL, "CELL.00",  PS_DATA_META, "",                    cell);

cell = psMetadataAlloc();
psMetadataAdd(cell, PS_LIST_TAIL, "EXTNAME",  PS_DATA_STR,  "",                    "CCD01");
psMetadataAdd(cell, PS_LIST_TAIL, "BIASNAME", PS_DATA_STR,  "",                    "BSEC-01");
psMetadataAdd(cell, PS_LIST_TAIL, "CHIP",     PS_DATA_STR,  "",                    "CHIP.01");
psMetadataAdd(md,   PS_LIST_TAIL, "CELL.01",  PS_DATA_META, "",                    cell);
```

The following code shows how to use the APIs to replace one of these values:

```
psMetadataAdd(md, PS_LIST_TAIL, "EXPTIME",  PS_DATA_F32 | PS_REPLACE,  "new exposure time (sec)",   2.05);
```

As a convenience to the user, the following type-specific functions are specified:

```
bool psMetadataAddStr(psMetadata* md, long location, const char* name, int format,
                      const char* comment, const char* value);
bool psMetadataAddS32(psMetadata* md, long location, const char* name, int format,
                      const char* comment, psS32 value);
bool psMetadataAddF32(psMetadata* md, long location, const char* name, int format,
                      const char* comment, psF32 value);
bool psMetadataAddF64(psMetadata* md, long location, const char* name, int format,
                      const char* comment, psF64 value);
bool psMetadataAddBool(psMetadata* md, long location, const char* name, int format,
                       const char* comment, bool value);
bool psMetadataAddPtr(psMetadata* md, long location, const char* name, psDataType type,
                      const char* comment, psPtr value);
```

Items may be removed from the metadata by specifying a key or a location in the list. For `psMetadataRemoveKey`, if the key matches a metadata item, the item is removed from the metadata and `true` is returned; otherwise, `false` is returned. If the key is not unique, then *all* items corresponding to the key are removed, and `true` is returned. For `psMetadataRemoveIndex`, the metadata item at the specified `location` is removed, if valid, and `true` is returned; otherwise the function returns `false`.

```
bool psMetadataRemoveKey(psMetadata *md, const char *key);
bool psMetadataRemoveIndex(psMetadata *md, int location);
```

Items may be found within the metadata by providing a key. In the event that the key is non-unique, the first item is returned.

```
psMetadataItem *psMetadataLookup(const psMetadata *md, const char *key);
```

Several utility functions are provided for simple cases. These functions perform the effort of casting the data to the appropriate type. The numerical functions shall return 0.0 if their key is not found. If the pointer value of `status` is not `NULL`, it is set to reflect the success or failure of the lookup.

```
psString psMetadataLookupStr(bool *status, const psMetadata *md, const char *key);
psS32 psMetadataLookupS32(bool *status, const psMetadata *md, const char *key);
psF32 psMetadataLookupF32(bool *status, const psMetadata *md, const char *key);
psF64 psMetadataLookupF64(bool *status, const psMetadata *md, const char *key);
bool psMetadataLookupBool(bool *status, const psMetadata *md, const char *key);
psPtr psMetadataLookupPtr(bool *status, const psMetadata *md, const char *key);
```

The following utility functions simplify further the selection of elements from the metadata. These functions convert the data associated with the supplied metadata item to the desired type if possible. Conversions between types are performed as needed, and default values are returned for the integer and floating point types if the data is not available (0 for int, `NaN` for float).

```
psBool psMetadataItemParseBool(const psMetadataItem *item);
psF32 psMetadataItemParseF32(const psMetadataItem *item);
psF64 psMetadataItemParseF64(const psMetadataItem *item);
psS8  psMetadataItemParseS8(const psMetadataItem *item);
```

```
psS16 psMetadataItemParseS16(const psMetadataItem *item);
psS32 psMetadataItemParseS32(const psMetadataItem *item);
psU8  psMetadataItemParseU8(const psMetadataItem *item);
psU16 psMetadataItemParseU16(const psMetadataItem *item);
psU32 psMetadataItemParseU32(const psMetadataItem *item);
psString psMetadataItemParseString(const psMetadataItem *item);
```

Items may be retrieved from the metadata by their entry position. The value of which specifies the desired entry in the fashion of `psList`.

```
psMetadataItem *psMetadataGet(const psMetadata *md, int location);
```

The metadata list component may be iterated over by using a `psMetadataIterator` in a fashion equivalent to the `psListIterator`:

```
typedef struct {
    psListIterator* iter;              ///< iterator for the psMetadata's psList
    regex_t* regex;                     ///< the subsetting regular expression
} psMetadataIterator;
```

The iterator may be set to a location in the `psMetadata` list, and the user may get the previous or next item in the list relative to that location. The iterators may be used to return the next key matching a POSIX `regex`, e.g., if the user only wants to iterate through `IPP.machines.sky` and doesn't want to bother with `IPP.machines.detector`. The iterator should iterate over every item in the metadata list, even those that are contained in a `PS_DATA_LIST`. The value `iterator` specifies the iterator to be used. In setting the iterator, the position of the iterator is defined by `location`, which follows the conventions of the `psList` iterators.

```
psMetadataIterator *psMetadataIteratorAlloc(psMetadata *md, long location, const char *regex);
bool psMetadataIteratorSet(psMetadataIterator *iterator, long location);
psMetadataItem *psMetadataGetAndIncrement(psMetadataIterator *iterator);
psMetadataItem *psMetadataGetAndDecrement(psMetadataIterator *iterator);
```

Metadata items may be printed to an open file descriptor based on a provided format. The format string is an sprintf format statement with exactly one % formatting command. If the metadata item type is a numeric type, this formatting command must also be numeric, and type conversion performed to the value to match the format type. If the metadata item type is a string, the formatting command must also be for a string (%s type of command). If the metadata type is any other data type, printing is not allowed.

```
bool psMetadataItemPrint(FILE *fd, const char *format, const psMetadataItem *item);
```

A complete metadata structure may be printed to the given file descriptor with the following command, where the level represents the desired indentation

```
bool psMetadataPrint(FILE *fd, const psMetadata *md, int level);
```

There will be occasions when we want to perform a deep copy of a `psMetadata`, for example, to generate a new and independent FITS header.

```
psMetadataItem *psMetadataItemCopy(const psMetadataItem *in);
psMetadata *psMetadataCopy(psMetadata *out, const psMetadata *in);
```

psMetadataItemCopy shall create a new copy of the input psMetadataItem. Since it is not feasible (at this time) to be able to copy every type, data of the standard numeric types plus PS_DATA_VECTOR, PS_DATA_TIME, PS_DATA_METADATA and PS_DATA_REGION shall be copied; other pointer types may simply be copied with a warning **for now (TBD)** .

psMetadataCopy shall create a new copy of all psMetadataItems in the in metadata, and place them in the out metadata, or a new psMetadata if out is NULL.

The following function copies a single metadata item, specified by the key from the in metadata to the out metadata:

```
bool psMetadataItemTransfer(psMetadata *out, const psMetadata *in, const char *key);
```

psMetadataItemCompare shall compare two metadata items, returning true if the names, and values are the same.

```
bool psMetadataItemCompare(const psMetadataItem *compare,
                           const psMetadataItem *template);
```

3.6.4 Configuration files

It will be necessary for the Pan-STARRS system, in order to load pre-defined settings, to parse a configuration file into a psMetadata structure. This shall be performed by the function psMetadataConfigParse, as described below.

```
psMetadata *psMetadataConfigParse(psMetadata *md, unsigned int *nFail, const char *filename, bool overwrite
```

Given a metadata container, md, and the name of a configuration file, filename, psMetadataConfigParse shall parse the configuration file, placing the contained key/type/value/comment quads into the metadata, and returning a pointer to the metadata structure. The number of lines that failed to parse is returned in nFail. Multiple specifications of a key that haven't been declared (see below) are overwritten if and only if overwrite is true. If the metadata container is NULL, it shall be allocated.

On error, the function shall return NULL.

It is also useful to be able to convert a psMetadata structure into the Configuration File format for debugging purposes and to enable persistent configuration.

```
psString psMetadataConfigFormat(psMetadata *md);
bool psMetadataConfigWrite(psMetadata *md, const char *filename);
```

The psMetadataConfigFormat function converts a psMetadata structure (including any nested psMetadata) into a Configuration File formatted string. A NULL shall be returned on error. The psMetadataConfigWrite behaves the same as psMetadataConfigFormat except that the string is written out to filename. false is returned on failure.

3.6.4.1 Comments

The configuration file shall consist of plain text with key/type/value/comment quads on separate lines. Blank lines, including those consisting solely of whitespace (both spaces and tabs), shall be ignored, as shall lines that commence with

the comment character (a hash mark, #), either immediately at the start of the line, or preceded by whitespace. The key/type/value/comment quads shall all lie on a single line, separated by whitespace.

The key shall be first, possibly preceded on the line by whitespace which should not form part of the key.

3.6.4.2 NULL values

The "value" of a quad may be declare to be undefined with the `NULL` keyword. `NULL` is allowed to co-exist with a "comment" and may be surrounded by whitespace. Any non-whitespace character will cause of the "value" to be interpreted as a string.

```
foo     STR     NULL    # string with a NULL value
bar     STR     NULL a  # string with a value of "NULL a"
```

3.6.4.3 Types

3.6.4.3.1 Scalar & Vector

Next, to assist the casting of the value, shall be a string identifying the type of the value, which shall correspond to one of the simple types supported in `psMetadata`: `STRING,BOOL,S32,F32,F64`; `STR` may be used to abbreviate `STRING`; valid time types are `UTC,UT1,TAI,TT`.

May, in the future, require more types, including U8,S16,C64. (TBD)

The value shall follow the type: strings may consist of multiple words, and shall have all leading and trailing whitespace removed; booleans shall simply be either `T` or `F`. Time type values will be in the ISO8601 compatible format of "YYYY-MM-DDTHH:MM:SS.sZ". When parsed, time types shall be represented as a `psTime` object.

Following the value may be an optional comment, preceded by a comment character (a hash mark, #), which in the case of a string value, serves to mark the end of the value, and for other types serves to identify the comment to the reader. Only one comment character may be present on any single line (i.e., neither strings nor comments are permitted to contain the comment character). The comment may consist of multiple words, and shall have leading and trailing whitespace removed.

One wrinkle is the specification of vectors. Keys for which the value is to be parsed as a vector shall be preceded immediately by a "vector symbol", which we choose to be the "at" sign, `@`. In this case, the type shall be interpreted as the type for the vector, which may be any of the signed or unsigned integer or floating point types (`U8,U16,U32,U64,S8,S16,S32,S32,S64,F32,F64`) but not the complex floating point types; and the value shall consist of multiple numbers, separated either by a comma or whitespace. These values shall populate a `psVector` of the appropriate type in the order in which they appear in the configuration file.

May add complex types, likely to be specified with values such as 1.23+4.56i in the future. (TBD)

May add null, Not-a-Number (NaN), de-normalized, underflow, overflow, and/or +/-infinity values for selected types. (TBD)

3.6.4.3.2 MULTI

An additional hurdle is the specification of keys that may be non-unique (such as the `COMMENT` keyword in a FITS header). These keys shall be specified in the configuration file as non-unique with a `MULTI` declaration. In the form `[keyword] MULTI`. No other data may be provided on this line, though a comment, preceded by the comment marker, is valid. A warning shall be produced when a key which has not been specified to be non-unique is repeated; in this case, the former value shall be overwritten if `overwrite` is `true`, otherwise the line shall be ignored and counted as one that could not be parsed. It should be noted that non-unique keys may be of mixed type (even the `TYPE` and `METADATA` complex types). For example:

```
comment     MULTI   # a comment
comment     STR     some string
comment     F32     1.23456
comment     BOOL    T
```

If a line does not conform to the rules laid out here, a warning shall be generated, it shall be ignored and counted as a line that could not be parsed. The total number of lines that were not able to be parsed (including those that were ignored because `overwrite` is `false`, and any other parsing problems, but not including blank lines and comment lines) shall be returned by the function in the argument `nFail`.

Here are some examples of lines of a valid configuration file:

```
Double     F64     1.23456789      # This is a comment
Float    F32 0.98765 # This is a comment too
String  STR This is the string that forms the value #comment

 # This is a comment line and is to be ignored
boolean     BOOL    T # The value of 'boolean' is 'true'

@primes U8  2,3 5 7,11,13 17 #   These are prime numbers

comment MULTI # The rest of this line is ignored, but 'comment' is set to be non-unique
comment STR This
comment STR     is
comment STR       a
comment STR        non-unique
comment STR                  key
Float F64 1.23456 # This generates a warning, and, if 'overwrite' is 'false', is ignored
```

Of course, a real configuration file should look much nicer to humans than the above example, but PSLib must be able to parse such ugly files.

3.6.4.4 Complex Types

3.6.4.4.1 TYPE

We support a modest tree structure by defining a reserved keyword `TYPE`. Any line in the config file which starts with the word `TYPE` shall be interpreted as defining a new valid type. The defined type name follows the word `TYPE`, and is in turn followed by an arbitrary number of words. These words are to be interpreted as the names of an embedded `psMetadata` entry, where the values are given on any line which (following the `TYPE` definition) employs the new type name. For example, a new type may be defined as:

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP
CELL.00   CELL   CCD00     BSEC-00  CHIP.00
CELL.01   CELL   CCD01     BSEC-01  CHIP.00
```

When `psMetadataConfigParse` encounters the `TYPE` line, it should construct a `psMetadata` container and fill it with `psMetadataItems` having the names `EXTNAME, BIASSEC, CHIP`, with type `PS_DATA_STR`, but data allocated. When it next encounters an entry of type `CELL`, it should then use the given name (e.g., `CELL.00`) for the `psMetadataItem`, and copy the `psMetadata` data onto the `psMetadataItem.data.md` entry, filling in the values from the rest of the line (`CCD00, BSEC-00, CHIP.00`). This hierarchical structure is illustrated in Figure 1.

3.6.4.4.2 METADATA

Another way to form a tree-like structure is to directly define a `psMetadata` entry using a sequence of successive lines to define the values of the `psMetadataItem` entries. The initial line defines the new `psMetadata` entry and its name. The following lines have the same format as the other metadata config file entries. The sequence is terminated with a line with a single word `END`. For example, a metadata entry may be defined as:

```
CELL      METADATA
 EXTNAME   STR   CCD00
 BIASSEC   STR   BSEC-00
 CHIP      STR   CHIP.00
 NCELL     S32   24
END
```

3.6.4.5 Scoping Rules

A simple set of "Scoping Rules" are required to properly parse a configuration file. "Scope" refers to the current "level" of METADATA that a statement appears in. Statements that are not contained in a nested METADATA are said to be in the "Top level scope". Each level of nested METADATA statements create a new "lower level scope".

- Variable names are unique only to the current level of scope.
- non-unique keywords (MULTI) apply only to the current scope. i.e. They are invalid in "higher" or "lower" level scopes.
- TYPE declarations apply only to the current scope.
- METADATA declarations must begin and end in the same scope. i.e. They may not be declared and end in two different nested METADATA and the same depth.

A series of test inputs is contained in §A.

3.7 Lookup Tables

Lookup tables store a variety of values indexed on a certain column. An example is for storing the difference between UT1 and UTC, and the polar motion vector as a function of date.

One of the key functionalities of a lookup table is to read data from an ordinary text file into an array of vectors. This functionality is generally useful, and so we specify a separate function that may be called independently:

```
psArray *psVectorsReadFromFile(const char *filename, const char *format);
```

psVectorsReadFromFile shall return an array of psVectors, read from the specified filename. The file shall be plain text, consisting of an identical number of columns on each line, with the values separated by whitespace. Lines commencing with a comment character (the pound sign, #) and blank lines shall be ignored. The format is a scanf-like format which specifies the number of columns in the file, as well as their types. The following formats shall be defined: \%d for psS32, \%ld for psS64, \%f for psF32, and \%lf for psF64. A star (*) in the format shall indicate that the column is to be skipped.

```
typedef struct {
    const char *filename;               ///< File from which data is to be read
    const char *format;                 ///< scanf-like format string for file
    long indexCol;                      ///< Column of the index vector (starting at zero)
    psVector *index;                    ///< Index values
    psArray *values;                    ///< Corresponding values: an array of vectors
    const double validFrom;             ///< Minimum index value for validity
    const double validTo;               ///< Maximum index value for validity
} psLookupTable;
```

filename shall specify the file from which the lookup table data is to be read. format shall contain a scanf-like format string specifying how the columns are to be interpreted (see psVectorsReadFromFile). indexCol shall specify the index of the column (with the first column having an index of zero) that will form the index values. index

shall contain the index values, which shall be sorted in increasing order. The `values` shall consist of an array of vectors, each of the same length as the `index` vector. The vectors (including the `index` and all vectors in the `values` array) may be any numerical type except complex types. The `validFrom` and `validTo` shall specify the range of valid values for the index; in most cases, these will simply be the first and last indices.

The constructor shall be:

```
psLookupTable *psLookupTableAlloc(const char *filename, ///< File from which to read
                                  const char *format,   ///< scanf-like format string
                                  long indexCol         ///< Column of the index vector (starting at zero)
                                  );
```

This function shall allocate a `psLookupTable`, and set the appropriate values, but it shall not read the lookup table. This is so that the lookup table can be specified at the initialisation of a program, but not read unless required.

The destructor shall free all the components.

```
psLookupTable *psLookupTableImport(psLookupTable *table,    ///< Lookup table into which to import
                                   const psArray *vectors,  ///< Array of vectors
                                   long indexCol            ///< Index of the index vector in the array of
                                   );
```

`psLookupTableImport` shall import an array of vectors into a `table`. If `table` is `NULL`, a new `psLookupTable` shall be allocated and returned. The array of `vectors`, which was likely generated by `psVectorsReadFromFile`, are imported by setting the `table->index` to the vector specified by the `indexCol`, and pointing the `table->values` array data to the remaining vectors in `vectors`. Reference counters for the vectors shall be incremented as appropriate. The `validFrom` and `validTo` members of the `table` shall be set to the first and last values in the index vector. If the `index` vector is not sorted in the file, the lookup table shall be sorted prior to the function returning.

```
long psLookupTableRead(psLookupTable *table);
```

`psLookupTableRead` combines `psVectorsReadFromFile` and `psLookupTableImport` to read the appropriate file and import the data into the extant `table`. If the input `table` has already been read from a file, the file shall be re-read, and the contents replaced. The function shall return the number of lines read (not including ignored lines).

Interpolation on a lookup table is performed by the following functions:

```
double psLookupTableInterpolate(const psLookupTable *table, double index,
                                long column);
psVector *psLookupTableInterpolateAll(const psLookupTable *table, double index);
```

Both functions shall interpolate the `table` at the provided `index`. For `psLookupTableInterpolate`, only the value in the specified `column` shall be calculated and returned. For `psLookupTableInterpolateAll`, all the values shall be calculated and returned as a `psVector`, the type of which shall be `PS_TYPE_F64`.

If the `index` is beyond the range of the `table`, `psLookupTableInterpolate` shall return `NaN`, and `psLookupTableInterpolateAll` shall return `NULL` — that is, no attempt is made at extrapolation.

4 Mathematical Structures

Throughout PSLib, we require a variety of structures which correspond to different mathematical data concepts. For example, we have a data structure which corresponds to one-dimensional arrays (vectors) of different data types (`int`, `float`, etc). We also have a data structure which corresponds to two-dimensional arrays (images or matrices), again with different data types for the individual elements.

A variety of functions perform operations which are equivalent for different data types of the same dimension, or may even be defined for different data types of different dimensions. For example, if we write the operation $x + y$, the operation is clearly defined regardless of whether the operands x and y are both zero dimensional (single numbers), one dimensional (vectors), two dimensional (images), etc. It is even reasonable to define the meaning of such an operation if the data dimensions do not match: if x is a scalar and y is an image, the natural operation is to add the value of x to every element of y; we can also define the meaning of the operation if x is a vector and y is a matrix. Nor does it matter mathematically that the element data types match; the sum of a float and an integer is a well-defined quantity. One constraint should be noted: the size of the elements in each dimension must match. For example, if x were a vector of 100 elements, but y were a vector of 1000 elements, the meaning of the operation $x + y$ is unclear. This type of operation should be invalid and should generate an error.

Given that some functions should be able to operate equivalently (or identically) on a wide range of data types, we define a mechanism which allows the C functions to accept a generic data type, and determine the type of the data on the basis of the data. Supported data types must be defined by a structure in which the first element is always of type `psMathType`:

```
typedef struct {
    psElemType type;                    ///< The type
    psDimen dimen;                      ///< The dimensionality
} psMathType;
```

where `psDimen dimen` defines the dimensionality of the data:

```
typedef enum {
    PS_DIMEN_SCALAR,                    ///< Scalar
    PS_DIMEN_VECTOR,                    ///< A vector
    PS_DIMEN_TRANSV,                    ///< A transposed vector
    PS_DIMEN_IMAGE,                     ///< An image (matrix)
    PS_DIMEN_OTHER                      ///< Not supported for arithmetic
} psDimen;
```

`psElemType type`, which defines the data type of each element, has already been defined

We discuss the application of `psMathType` in more detail in section 6.8.

4.1 Scalars

We define a basic scalar data type which includes the type information. This structure allows scalars to be used in functions which interpret the data type from the structure when deciding how to perform an operation. The basic scalar structure is:

```
typedef struct {
    psMathType type;                    ///< data type information
    union {
        psS8   S8;                      ///< bye value entry
```

```
        psS16 S16;                      ///< short int value entry
        psS32 S32;                      ///< int value entry
        psS64 S64;                      ///< long int value entry
        psU8   U8;                      ///< unsigned byte value entry
        psU16 U16;                      ///< unsigned short int value entry
        psU32 U32;                      ///< unsigned int value entry
        psU64 U64;                      ///< unsigned long int value entry
        psF32 F32;                      ///< float value entry
        psF64 F64;                      ///< double value entry
        psC32 C32;                      ///< float complex value entry
        psC64 C64;                      ///< double complex value entry
    } data;
} psScalar;
```

In addition, we specify two functions for working with `psScalar` data:

```
psScalar *psScalarAlloc(double complex value, psElemType type);
psScalar *psScalarCopy(const psScalar *value);
```

The first creates a `psMathType`-ed structure from a constant value, casting it as appropriate based on the `type`. The second copies the provided `psScalar` value. This latter function is necessary to keep a copy of an existing `psScalar` value, since `psBinaryOp` and `psUnaryOp` are required to free incoming `psScalar` data (see §6.8).

4.2 Vectors

We require several related types of basic one-dimensional arrays: arrays of values of type `int`, `float`, `double`, `float complex`, and `double complex`. We have defined a single structure, `psVector` to represent these concepts:

```
typedef struct {
    psMathType type;                    ///< vector data type and dimension
    long n;                             ///< size of vector
    const long nalloc;                  ///< allocated data block
    union {
        psS8  *S8;                      ///< Pointers to byte data
        psS16 *S16;                     ///< Pointers to short-integer data
        psS32 *S32;                     ///< Pointers to integer data
        psS64 *S64;                     ///< Pointers to long-integer data
        psU8  *U8;                      ///< Pointers to unsigned-byte data
        psU16 *U16;                     ///< Pointers to unsigned-short-integer data
        psU32 *U32;                     ///< Pointers to unsigned-integer data
        psU64 *U64;                     ///< Pointers to unsigned-long-integer data
        psF32 *F32;                     ///< Pointers to floating-point data
        psF64 *F64;                     ///< Pointers to double-precision data
        psC32 *C32;                     ///< Pointers to complex floating-point data
        psC64 *C64;                     ///< Pointers to complex double-precision data
    } data;
    void *lock;                         ///< Lock for thread safety
} psVector;
```

In this structure, the argument `n` is the length of the array (the number of elements); `nalloc` is the number of elements allocated (`nalloc` $\geq$ `n`). The allocated memory is available in the union `data` which consists of pointers to each of the defined primitive data types. Note the parallelism in the names of the types, union elements, and the `psElemType` names. This parallelism allows us to use automatic construction mechanisms effectively. The data type is defined by the first element, `psMathType`. The structure is associated with a constructor and reallocator:

```
psVector *psVectorAlloc(long nalloc, psElemType type);
psVector *psVectorRealloc(psVector *vector, long nalloc);
psVector *psVectorRecycle(psVector *vector, long nalloc, psElemType type);
```

In these functions, `nalloc` is the number of elements to allocate. For `psVectorAlloc`, the value of `psVector.n` is initially set to zero. Users may choose to restrict the data range after the `psVectorAlloc` function is called. For `psVectorRealloc`, if the value of `nalloc` is smaller than the current value of `psVector.n`, then `psVector.n` is set to `nalloc`, the array is adjusted down to match `nalloc`, and the extra elements are lost. If `nalloc` is larger than the current value of `psVector.n`, `psVector.n` is left intact. If the value of `vector` is `NULL`, then `psVectorRealloc` must generate an error.

`psVectorRecycle` shall recycle the input `vector`, such that the output `psVector` matches the length required for `nalloc` elements of the specified `type`. In the event that the input `vector` is `NULL`, a new `psVector` shall be allocated and returned.

```
long psVectorLength(const psVector *vector);
```

This function returns the length of the vector (`psVector.n`).

```
psVector *psVectorExtend(psVector *vector, long delta, long nExtend);
```

This function increments `psVector.n` (the number of elements in the vector) by `nExtend`. If the current length of the vector plus *twice* the number of new elements (`nExtend`) is greater than the allocated space, an additional `delta` elements are allocated. The function shall generate an error if `nExtend` is negative. If the value of `delta` is less than 1, 10 shall be used.

Here is an example of how `psVectorExtend` is used to automatically increment the vector length.

```
  // create data vector
  psVector *y = psVectorAlloc(100, PS_TYPE_F32);
  y->n = 0;
  for (int i = 0; i < 1000; i++) {
    y->data.F32[y->n + 0] = 2*i;
    y->data.F32[y->n + 1] = 2*i;
    y->data.F32[y->n + 2] = 2*i;
    y = psVectorExtend (y, 100, 3);
    // increments n by 3, extends length by 100 if needed
  }
```

Note that the specification that the allocation always be greater than the number of elements by twice the number of new elements implies that there will be room on the next loop for `nExtend` new elements, as in this example.

```
psVector *psVectorCopy(psVector *output, const psVector *input, psElemType type);
```

`psVectorCopy` shall copy the elements in the `input` vector to the `output` vector, casting to the specified `type`. In the event that the `output` is `NULL`, a new `psVector` shall be allocated. The returned `psVector` shall be of the given `type`.

```
bool psVectorInit(psVector *vector, ...);
```

`psVectorInit` shall initialize the vector with the given value. The input data is cast to match the vector datatype, allowing for integers to be preserved.

```
bool psVectorSet(psVector *input, long position, double complex value);
double complex psVectorGet(const psVector *input, long position);
```

These accessor functions are provided as a convenience to the user. `psVectorSet` sets the value of the `input` vector at the specified `position` to `value` (appropriately cast), returning `true` if successful. `psVectorGet` returns the value of the `input` vector at the specified `position`. A negative `position` means index from the end.

```
psVector *psVectorCreate(psVector *input, double lower, double upper, double delta, psElemType type);
```

This function creates a new vector, or reallocates the provided vector if `input` is not `NULL`. The created vector consists of the data range starting at `lower`, running to `upper`, in steps of `delta`. The upper-end value is *exclusive*; the sequence is equivalent to `for (x = lower; x < upper; x += delta)`.

4.3 Images

The most important data product produced by the telescope is an image. The simplest image is a 2-D collection of pixels, each with some value. We require a basic image data type:

```
typedef struct psImage {
    const psMathType type;              ///< image data type and dimension
    const int numCols;                  ///< Number of columns in image
    const int numRows;                  ///< Number of rows in image.
    int col0;                           ///< Column position relative to parent.
    int row0;                           ///< Row position relative to parent.
    union {
        psS8  **S8;                     ///< Pointers to char data
        psS16 **S16;                    ///< Pointers to short-integer data
        psS32 **S32;                    ///< Pointers to integer data
        psS64 **S64;                    ///< Pointers to long-integer data
        psU8  **U8;                     ///< Pointers to unsigned-char data
        psU16 **U16;                    ///< Pointers to unsigned-short-integer data
        psU32 **U32;                    ///< Pointers to unsigned-integer data
        psU64 **U64;                    ///< Pointers to unsigned-long-integer data
        psF32 **F32;                    ///< Pointers to floating-point data
        psF64 **F64;                    ///< Pointers to double-precision data
        psC32 **C32;                    ///< Pointers to complex floating-point data
        psC64 **C64;                    ///< Pointers to complex double-precision data
        psPtr *V;                       ///< Pointers to untyped data
    } data;
    const struct psImage *parent;       ///< parent, if a subimage
    psPtr p_rawDataBuffer;              ///< Raw data; private
    psArray *children;                  ///< children of this region
    void *lock;                         ///< Lock for thread safety
} psImage;
```

This structure represents an image consisting of a 2-D array of pixels. The size of this array is given by the elements `(numRows, numCols)`. The data type of the pixel is defined in the `psMathType type` entry (specifically, the

psElemType member, type; see §6.8). (N.B. that for FITS images, these values are restricted to the datatypes equivalent to the valid BITPIX values 8, 16, 32, -32, -64). The image represented in the data structure may represent a subset of the pixels in a complete array, in which case the image is considered to be the child of that parent array. The offset of the (0,0) pixel in this array relative to the parent array is given by the elements (col0,row0): col0 is the starting column number in the parent image while row0 is the starting row number. The structure may include references to subrasters (children) and/or to a containing array (parent). Unless this image is a child of another image (represents a subset of the pixels of another image), the image data is allocated in a contiguous block (data.V).

4.3.1 Support Functions

We define the following supporting functions for images, which are valid for data types psS8, psS16, psU8, psU16, psF32, psF64, psC32, psC64.

```
psImage *psImageAlloc(int numCols, int numRows, psElemType type);
```

Create an image of a specified numCols, numRows, and data type. This function must allow any of the valid image data types and not restrict to the valid FITS BITPIX types. The image dimensionality must be 2.

```
psImage* psImageRecycle(
    psImage* old,                       ///< the psImage to recycle by resizing image buffer
    int numCols,                        ///< the desired number of columns in image
    int numRows,                        ///< the desired number of rows in image
    const psElemType type               ///< the desired datatype of the image
);
```

psImageRecycle shall recycle the input old image, such that the output psImage matches the specified size (numCols×numRows) and type. In the event that the input old image is NULL, a new psImage shall be allocated and returned.

```
int psImageFreeChildren(psImage *image);
```

psImageFreeChildren shall free all child images of the given image.

```
bool psImageInit(psImage *image, ...);
```

psImageInit shall initialize the image with the given value. The input data is cast to match the image datatype, allowing for integers to be preserved.

```
bool psImageSet(psImage *image, int x, int y, double complex value);
double complex psImageGet(const psImage *image, int x, int y);
```

These accessor functions are provided as a convenience to the user. psImageSet sets the value of the image at the specified x,y position to value (appropriately cast), returning true if successful. psImageGet returns the value of the image at the specified x,y position. A negative value for the x,y position means index from the end.

4.3.2 Image Regions

In many places, we need to refer to a rectangular area. We define a structure to represent a rectangle:

```
typedef struct {
    float x0;
    float x1;
    float y0;
    float y1;
} psRegion;
```

This structure is used in psLib as an abbreviation for the four floats, and is defined statically. Functions which accept or return a `psRegion` shall do so by value, not by pointer.

In the limited cases where we wish to store a `psRegion` on one of the containers, we need an attached `psMemBlock`, which can be obtained by calling `psRegionAlloc`:

```
psRegion *psRegionAlloc(float x0, float x1, float y0, float y1);
```

All functions which use a `psRegion` must interpret the definition of $(x0, y0)$ and $(x1, y1)$ in the same way. The coordinate $(x0, y0)$ defines the starting pixel in the region. The coordinate $(x1, y1)$ defines the outer bound of the region, and are NOT included in the region. The size of the region is $(x1 - x0, y1 - y1)$. If either $x1$ or $y1$ is less than or equal to 0, the value is added to the image dimensions (e.g., $Nx + x1$). Thus a region `[0:0,0:0]` refers to the full image array, while `[0:-10,0:-20]` refers to the entire image, minus the last 10 columns and the last 20 rows.

We define two functions to set and return the value of a `psRegion`. The first defines the region by the corner coordinates. The second function converts the IRAF description region in the form `[x0:x1,y0:y1]`, used for header entries such as `BIASSEC`, into the corresponding `psRegion` structure (any values that do not parse correctly shall be returned as `NaN`). We also define a function that converts a `psRegion` to the corresponding IRAF description.

```
psRegion psRegionSet(float x0, float x1, float y0, float y1);
psRegion psRegionFromString(const char *region);
psString psRegionToString(const psRegion region);
```

Note that regions specified by strings are in the FITS standard. It is the responsibility of `psRegionFromString` and `psRegionToString` to convert between the PS standard (0 means first pixel; upper value is not included, but lower is) and the FITS standard (1 means first pixel; lower and upper values are included), which simply involves subtracting one from `x0` and `y0` when going from a string representation to a `psRegion`. A `NULL` string is allowed and is equivalent to the default region (`0:0,0:0`).

```
psRegion psRegionForImage(psImage *image, psRegion in);
```

An image region defined with negative upper limits may be rationalized for the bounds of a specific image with `psRegionForImage`. The output of this function is a region with negative upper limits replaced by their corrected value appropriate to the given image. In addition, the lower and upper limits are forced to lie within the bounds of the image. If the lower-limit coordinates are less than the lower bound of the image, they are limited to the lower bound of the image. Conversely, if the upper-limit coordinates are greater than the upper bound of the image, they are truncated to define only valid pixels. If the lower-limit coordinates are greater than the upper bounds of the image, or the upper-limit coorindates are less than the lower bounds of the image, the coordinates should saturate on those limits. The output of this

function is always a valid region, though it may define an area of 0 pixels. If `image` is a subimage, the input and output region coordinates refer to the parent pixel coordinates. The only exception to this statement is that the negative limits should be applied to the upper limits of the subimage, not the parent. Thus, if we have an input subimage with `col0,row0` of (10,20), and `numCols,numRows` of 1000,1000 (implying parent image dimensions of at least 1010,1020), we would have the following conversions:

```
(50:100,50:100) -> (50:100,50:100)   : no change (region within image)
(0:0,0:0)       -> (10:1010,20:1020) : upper and lower limits constrained
(5:-5,5:-5)     -> (10:1005,20:1015)
(5:1020,5:1020) -> (10:1010,20:1020)
```

```
psRegion psRegionForSquare(double x, double y, double radius);
```

This utility function defines a `psRegion` corresponding to the square with center at coordinate `x,y` and with coderadius. The width of the square is thus `2radius + 1`.

The following fucntions provide tests of the validity of `regions`

```
bool psRegionIsNaN(psRegion region);
```

5 Input/Output

5.1 XML Functions

Within Pan-STARRS, we will use XML documents as a transport mechanism to carry data between programs and between IPP and other subsystems. Configuration information may be stored as well as XML documents, in addition to the text format discussed in the discussion on Metadata. XML is an extremely variable document format, and it is not currently the intention of PSLib to provide a complete PSLib version of XML operations. Rather, a limited number of operations are defined to convert specific data structures to an appropriate XML document. To maximize the simplicity of the XML APIs, we will use the convention that a single XML document to be parsed by PSLib shall contain only a single data structure. Each of the XML APIs therefore take as input a reference to a complete XML document and return a PSLib data structure, or take a PSLib data structure and return a complete XML document.

We start by defining a PSLib wrapper type which is a pointer to an XML document in memory. We wrap the `libxml2` version of an XML document pointer for now:

```
typedef xmlDocPtr psXMLDoc;


psXMLDoc *psXMLDocAlloc(void);
```

The next pair of functions convert a `psMetadata` data structure to a complete `psXMLDoc` (in memory) and vice versa:

```
psXMLDoc *psMetadataToXMLDoc(const psMetadata *md);
psMetadata *psXMLDocToMetadata(const psXMLDoc *doc);
```

The next pair of functions loads the data in a named file into a complete `psXMLDoc` (in memory) and write out the `psXMLDoc` to a named file:

```
psXMLDoc *psXMLParseFile(const char *filename);
bool psXMLDocToFile(const psXMLDoc *doc, const char *filename);
```

The next pair of functions accepts a block of memory and parses it into a complete `psXMLDoc` (also in memory), and vice versa:

```
psXMLDoc *psXMLParseMem(const char *buffer, int size);
bool psXMLDocToMem(const psXMLDoc *doc, char *buffer);
```

The next pair of functions read from and write to a file descriptor. The first converts the incoming data to a complete `psXMLDoc` (also in memory), the second writes the `psXMLDoc` to the file descriptor:

```
psXMLDoc *psXMLParseFD(int fd);
bool psXMLDocToFD(const psXMLDoc *doc, int fd);
```

5.2 Database Functions

Many of the applications that PSLib will be used for will require access to a simple relational database. PSLib includes generic database-independent interface mechanisms as part of its API set. The most important aspect of PSLib's database

support is to abstract as much database specific complexity as is feasible. As almost all RDBMS provide at least a simple transactional model, commit and rollback support should be provided.

Currently, only support for MySQL 4.1.x is required but other backends may be added as options in the future. As a particular example which has implications for the database interaction model, support for SQLite may be required in the future. Currently, the choice of backend database interface may be made as a compile option. Details of the specified APIs in the discussion below refer to the relevant MySQL functions.

Database errors must be trapped and placed onto the psError stack. The complete error message should be retrieved with the database's error function.

5.2.1 Managing the Database Connection

We specify a database handle which carries the information about the database connection:

```
typedef struct {
    void *mysql;
} psDB;
```

The following collection of functions provides basic database functionality:

```
psDB *psDBInit(const char *host, const char *user, const char *passwd, const char *dbname);
void psDBCleanup(psDB *dbh);
bool psDBCreate(psDB *dbh, const char *dbname);
bool psDBChange(psDB *dbh, const char *dbname);
bool psDBDrop(psDB *dbh, const char *dbname);
```

For MySQL support, `psDBInit` wraps `mysql_init` and `mysql_real_connect` in order to initialize a psDB structure and establish a database connection. A null pointer should be returned on failure.

When implementing support for SQLite, or other DB which is purely file-based, the `host`, `user`, and `passwd` arguments would be ignored while `dbname` would specify the path to the SQLite db file.

`psDBCleanup` shall wrap `mysql_close`. `psDBCreate` shall wrap `mysql_create_db`. `psDBChange` shall wrap `mysql_select_db`. `psDBDrop` shall wrap `mysql_drop_db`.

5.2.2 Interacting with Database Tables

The functions in this section perform high level interactions with the database tables. All of them should behave "atomically" with respect to the state of the database. Specifically, all interactions with the database should be done as a part of a transaction that is rolled-back on failure and committed only after all queries used by the API have been run. In general, this API set attempts to treat a database table as a 2D matrix where columns can be represented by a `psVector` and rows as a `psMetadata` type. A `psMetadata` collection is also used to define the columns of a table and as part of the query restrictions.

```
bool psDBCreateTable(psDB *dbh, const char *tableName, const psMetadata *md);
```

This function generates and executes the SQL needed to create a table named `tableName`, with the column names and datatypes as described in `md`. Each data item in the `psMetadata` collection represents a single table field. The name of the field is given by the name of the `psMetadataItem` and the data type is given by the `psMetadataItem.type` entry. A lookup table should be used to convert from PSLib types into MySQL compatible SQL data types. For example, a `PS_DATA_STR` would map to an SQL99 varchar. If the value of `type` is `PS_DATA_STR` then the `psMetadataItem.data` element is set to a string with the length for the field written as a text string. The value of the `psMetadataItem.data` element is unused for the `PS_DATA_PRIMITIVE` types. Other metadata types beyond `PS_DATA_STR` and `PS_DATA_PRIMITIVE` are not allowed in a table definition metadata collection.

Database indexes can be specified by setting the `comment` field to "`Primary Key`" or "`Key`". "Auto-incrementing" columns may be specified by setting the `comment` field to "`AUTO_INCREMENT`". Indexes and auto-increments maybe be combined in the same `comment`. They must be separated by optional whitespace, a comma, and then more optional whitespace. The `comment` field is otherwise ignored.

```
bool psDBDropTable(psDB *dbh, const char *tableName);
```

This function deletes the specified table.

```
bool p_psDBRunQuery(psDB *dbh, const char *format, ...);
```

This function will execute a string as a raw SQL query. `format` is a `printf` style formatting code to be implimented with `vsprintf()`. No additional processing of the string or abstraction of the underlying database's SQL dialect is provided.

```
psArray *psDBSelectColumn(psDB *dbh, const char *tableName, const char *col, unsigned long long limit);
psVector *psDBSelectColumnNum(psDB *dbh, const char *tableName, const char *col,
                              psElemType type, unsigned long long limit);
```

These functions generates and executes the SQL needed to select an entire column from a table or up to `limit` rows from it. If `limit` is 0, the entire range is returned. The database response is processed and a `psArray` of strings is returned. The Num version of the function returns the data in a `psVector`, data cast to `type`. It returns an error (NULL) if the requested field is not a numerical type.

```
psArray *psDBSelectRows(psDB *dbh, const char *tableName, const psMetadata *where, unsigned long long limit
```

This function returns rows from the specified table which match the restrictions given by `where`. The restrictions are specified as field / value pairs. The `psMetadata` collection where must consist of valid database fields, though the database query checking functions may be used to validate the fields as part of the query. If `where` is `NULL`, then there are no restrictions on the rows selected. The selected rows are returned as a `psArray` of `psMetadata` values, one per row.

```
bool psDBInsertOneRow(psDB *dbh, const char *tableName, const psMetadata *row);
```

Insert the data from `row` into `tableName`. It should be noted in the API reference that if fields are specified in `row` that do not exist in `tablename`, the insert will fail.

```
bool psDBInsertRows(psDB *dbh, const char *tableName, const psArray *rowSet);
```

Similar to `psDBInsertOneRow()`, this function inserts many rows at once and is atomic for the complete set of rows.

```
psArray *psDBDumpRows(psDB *dbh, const char *tableName);
```

Fetch all rows as an psArray of psMetadata.

```
psMetadata *psDBDumpCols(psDB *dbh, const char *tableName);
```

Fetch all columns, as either a psVector or a psArray depending on whether or not the column is numeric, and return them in a psMetadata structure where psMetadataItem.name contains the column's name.

```
long psDBUpdateRows(psDB *dbh, const char *tableName, const psMetadata *where,
                    psMetadata *values);
```

Update the columns contained in `values` in the row(s) that have a field with the value indicated by `where` (note that this is only allows very limited use of SQL99's "where" semantics). The number of rows modified is returned. A negative value is return to indicate an error. If there are multiple psMetadataItems in `where` then each item should be considered as an additional constraint. e.g. "where foo = x and where bar = y"

```
long psDBDeleteRows(psDB *dbh, const char *tableName, const psMetadata *where, unsigned long long limit);
```

Delete the rows that are matched by `where` using the same semantics for `where` as in psDBUpdateRow(). `limit` specifies the maximum number of rows that may be deleted per invocation. A negative value is returned to indicate an error.

```
long psDBLastInsertID(
    psDB *dbh                          ///< Database handle
);
```

Returns the last value created in an "auto-increment" field.

5.2.3 Transaction Control Database Functions

By default the database functions shall behave as if each operation effects a permanent change to the database. An API is provided to change this behavior such that operations may be grouped together into a "transaction" that is atomic.

```
bool psDBExplicitTrans(
    psDB *dbh,                          ///< Database handle
    bool mode                           ///< transactions enable/disable
);
```

Enable/Disable explicit transactions. When enabled `psDBCommit()` must be called to make changes to the database's state persistent.

```
bool psDBTransaction(
    psDB *dbh                           ///< Database handle
);
```

Start a new transaction.

```
bool psDBCommit(
    psDB *dbh                           ///< Database handle
);
```

Commit the current transaction.

```
bool psDBRollback(
    psDB *dbh                           ///< Database handle
);
```

Undo the current transaction.

5.2.4 Low Level Database Functions

This collection of functions is primarily intended for us internally by the other database functions. However, in certain cases their direct use may be needed to express semantics that are not supported by the public database interface.

```
long p_psDBRunQueryPrepared(
    psDB *dbh,                          ///< Database handle
    const psArray *rowSet,              ///< row data as psArray of psMetadata
    const char *format,                 ///< SQL string to execute
    ...
);
```

`p_psDBRunQueryPrepared()` is similar to `p_psDBRunQuery()` except that `format` is expected to be a query string with value "place holders" in it. This allows the data in `rowSet` is inserted into the database via the use of a more efficent "prepared query".

```
psArray *p_psDBFetchResult(
    psDB *dbh                          ///< Database handle
);
```

`p_psDBFetchResult()` is the inverse operation of `p_psDBRunQuery`. It shall load the latest result set from the database into an `psArray` and return it.

5.3 FITS I/O Functions

We need a variety of I/O functions between the disk and certain of our PSLib data structures. We need the ability to access FITS headers, images and tables (both ASCII and Binary). We define here the FITS I/O functions, all of which are currently specified as wrappers to functions within CFITSIO. CFITSIO provides a wide range of utilities which we do not feel are particularly appropriate as part of a generic I/O library, such as assumptions about names which change the data interpretation, etc. We are defining our calls to avoid the hidden 'features'. The CFITSIO functions which are wrapped should in general be the most basic versions.

```
typedef struct {
    fitsfile *fd;                // cfitsio structure
    bool writable;               // Is the file writable?
} psFits;
```

We begin by defining a datatype to wrap the CFITSIO `fitsfile` structure. This is necessary to allow repeated access to the data in a file without multiple open commands (which are expensive). Write operations are only permitted if `writable` is `true`.

5.3.1 FITS File Manipulations

```
psFits *psFitsOpen(const char *filename, const char *mode);
```

Opens a FITS file and positions the pointer to the PHU. The file is opened in the requested `mode`, which may be one of `r` (read only) `r+` (read and write), `rw` (alias for `r+`) or `w` (create new file for writing).

```
bool psFitsClose(psFits *fits);
```

Closes a FITS file.

```
bool psFitsMoveExtName(const psFits *fits, const char *extname);
```

Positions the pointer to the beginning of the specified `extname`. If the `extname` does not exist, the function shall fail.

```
bool psFitsMoveExtNum(const psFits* fits, int extnum, bool relative);
```

Moves the pointer to the beginning of the specified HDU number. If `relative` is TRUE, `extnum` represents the number of HDUs relative to the current HDU. The PHU is entry number 0, while the extended data segments start at number 1.

```
bool psFitsMoveLast(psFits *fits);
```

Moves the pointer to the last extension in the file.

```
int psFitsGetExtNum(const psFits* fits);
```

Returns the current HDU number (i.e., file position).

```
int psFitsGetSize(const psFits* fits);
```

Returns the number of HDUs in the file.

```
bool psFitsDeleteExtNum(psFits *fits, int extnum, bool relative);
bool psFitsDeleteExtName(psFits *fits, const char *extname);
```

These functions are similar to `psFitsMoveExtNum` and `psFitsMoveExtName` except that they delete the specified extension. The file pointer is left pointing to where the extension was before deletion.

```
bool psFitsTruncate(psFits *fits);
```

Deletes all extensions after the position of the file pointer.

```
typedef enum {
    PS_FITS_TYPE_NONE           = -1,
    PS_FITS_TYPE_IMAGE          = IMAGE_HDU,
    PS_FITS_TYPE_BINARY_TABLE   = BINARY_TBL,
    PS_FITS_TYPE_ASCII_TABLE    = ASCII_TBL,
    PS_FITS_TYPE_ANY            = ANY_HDU
} psFitsType;

psFitsType psFitsGetExtType(const psFits* fits);
```

Gets the current HDU's type (table or image).

```
psString psFitsGetExtName(const psFits* fits);
bool psFitsSetExtName(psFits* fits, const char* name);
```

`psFitsGetExtName` shall return the name of the current extension for the given `fits` file (as specified by the `EXTNAME` header). `psFitsSetExtName` shall change the name of the current extension for the given `fits` file to `name`, returning `true` upon success and `false` otherwise.

5.3.2 FITS Header I/O Functions

```
psMetadata *psFitsReadHeader(psMetadata *out, const psFits *fits);
```

Read header data into a `psMetadata` structure. The data is read from the current HDU pointed at by the `psFits *fits` entry. If `out` is `NULL`, a new psMetadata is created.

```
psMetadata *psFitsReadHeaderSet(psMetadata *out, const psFits *fits);
```

Load a complete set of headers from the `psFits` file pointer. This function loads the headers from all extensions into a `psMetadata` collection, each entry of which is a pointer to a `psMetadata` structure containing the header data. The metadata entry names are the `EXTNAME` values for each header (with the value of `PHU` for the primary header unit). At the start of the operation, the file pointer is rewound to the beginning of the file. At the end, it is positioned where it started when the function was called. If `out` is `NULL`, a new psMetadata is created.

```
bool psFitsWriteHeader(psFits *fits, const psMetadata *output);
```

Write metadata into the header of a FITS image file. The header is written at the current HDU.

```
bool psFitsWriteBlank(psFits *fits, const psMetadata *output);
```

This function creates a header in the `fits` file for a 0 length image. The resulting header shall have `NAXIS = 0`. Any `NAXISi` elements present in the header shall be maintained as reference data.

```
bool psFitsHeaderValidate(psMetadata *header);
```

`psFitsHeaderValidate` shall validate the supplied `header` so that it is in compliance to the FITS standard for header keyword names and types. This involves scanning for types that cannot be represented in FITS, and changing the keywords to conform to the FITS standard where possible by using the `HIERARCH` keyword. If the resulting `header` conforms to the FITS standard, the function shall return `true`; otherwise the function shall return `false`.

```
bool psFitsIsImage(psMetadata *header);
bool psFitsIsTable(psMetadata *header);
```

`psFitsIsImage` shall return `true` if the FITS `header` corresponds to that of a FITS image. `psFitsIsTable` shall return `true` if the FITS `header` corresponds to that of a FITS table.

5.3.3 FITS Image I/O Functions

```
psImage *psFitsReadImage(const psFits *fits, psRegion region, int z);
```

Read an image or subimage from the `psFits` file pointer. This function is a wrapper to the CFITSIO library function. The input parameters allow a full image or a subimage to be read. The region to be read is specified by `region`. A negative value for either of `region.x1` or `region.y1` specifies the size of the region to be read counting down from the end of the array.

If the native image is a cube, the value of z specifies the requested slice of the image. This function must call `psError` and return `NULL` if any of the specified parameters are out of range for the data in the image file, or if the image on disk is zero- or one-dimensional. This function need only read images of the native FITS image types (`psU8`, `psS16`, `psS32`, `psF32`, `psF64`). The user is expected to convert the data type as needed with `psImageCopy`.

```
bool psFitsUpdateImage(psFits *fits, const psImage *input, int x0, int y0, int z);
```

Write an image section to the open `psFits` file pointer. This operation may write a portion of an image over the existing bytes of an existing image, starting at `x0,y0` in the `fits` image. Note that the origin of the `input` must be (0,0), not that of any parent (i.e., `input->col0,input->row0`)). Care must be taken to interpret region which specifies the output pixels to be written / over-written. If the combination of `x0,y0` and the size of `psImage *input` implies writing pixels outside the existing data area of the image, the function shall return an error (ie, if `x0 + image.nx >= NAXIS1`, `y0 + image.ny >= NAXIS2`, or `z >= NAXIS3`). This function will only write images of the native FITS image types (`psU8`, `psS16`, `psS32`, `psF32`, `psF64`). The user is expected to convert the data type as needed with `psImageCopy`. The return value must be 0 for a successful operation and 1 for an error.

```
bool psFitsWriteImage(psFits *fits, const psMetadata *header, const psImage *input, int depth,
                      const char *extname);
```

Create a new image based on the dimensions specified for the image and the requested depth. The header and image data segments are written at the end of the file. This function will only write images of the native FITS image types (`psU8`, `psS16`, `psS32`, `psF32`, `psF64`). The user is expected to convert the data type as needed with `psImageCopy`. The return value must be 0 for a successful operation and 1 for an error.

```
bool psFitsInsertImage(psFits *fits, const psMetadata *header, const psImage *input, int depth,
                       const char *extname, bool after);
```

`psFitsInsertImage` behaves in the same manner as `psFitsWriteImage`, except that the extension is inserted according to the value of the boolean `after`. If `after` is `true`, then the extension is inserted after the current `psFits` pointer; otherwise the extension is inserted before the current `psFits` pointer. **The user should beware that this is potentially a very expensive operation in terms of time, since the entire file following the inserted extension must be rewritten.**

We also provide the following functions that use a `psArray` of `psImages`, so that the user need not worry about updating `NAXIS` headers:

```
psArray *psFitsReadImageCube(const psFits *fits, psRegion region);
bool psFitsUpdateImageCube(psFits *fits, const psArray *input, int x0, int y0);
bool psFitsWriteImageCube(psFits *fits, psMetadata *header, const psArray
*input, const char *extname);
bool psFitsInsertImageCube(psFits *fits, psMetadata *header, const psArray
*input, const char *extname, bool after);
```

5.3.4 FITS Table I/O Functions

```
psMetadata *psFitsReadTableRow(const psFits *fits, int row);
```

This function reads a single row of the table in the extension pointed at by the `psFits` file pointer. The row number to be read is given by `row`. The result is returned as a `psMetadata` collection with elements of the appropriate types and keys corresponding to the table column names. The function must apply the needed byte-swapping on the data in the row based on the description of the table data in the table header. **we may need to be more flexible here: if we call this function repeatedly, it would be more efficient to pass the corresponding header or keep it somewhere (and the file pointer location, for that matter). (TBR)**

```
psPtr psFitsReadTableRowRaw(size_t *size, const psFits *fits, int row);
```

This function reads a single row of the table in the extension pointed at by the `psFits` file pointer. The row number to be read is given by `row`. The result is returned as collection of `size` bytes allocated by the function. The function must apply the needed byte-swapping on the data in the row based on the description of the table data in the table header. **we may need to be more flexible here: if we call this function repeatedly, it would be more efficient to pass the corresponding header or keep it somewhere (and the file pointer location, for that matter). (TBR)**

```
psArray *psFitsReadTableColumn(const psFits *fits, const char *colname);
```

This function reads a single column of the table in the extension pointed at by the `psFits` file pointer. The column is specified by the FITS table column key given by `row`. The result is returned as a `psArray`, with the data from one row of the table column per array element.

```
psVector *psFitsReadTableColumnNum(const psFits *fits, const char *colname);
```

This function reads a single column of the table in the extension pointed at by the `psFits` file pointer. The column is specified by the FITS table column key given by `row` and must be of a numeric data type. The result is returned as a `psVector` of the appropriate data type, with the data from one row of the table column per array element.

```
psArray *psFitsReadTableRaw(size_t *size, const psFits *fits);
```

This function reads the entire data block from a table into the a `psArray`, with one element of the array per row. The number of bytes per row is returned in `size`. The function must apply the needed byte-swapping on the data in each row based on the description of the table data in the table header.

```
psArray *psFitsReadTable(const psFits *fits);
```

This function reads the entire data block from a table into the a `psArray`, with one element of the array per row. Each row is stored as a `psMetadata` collection as described above for `psFitsReadTableRow`.

```
bool psFitsWriteTable(psFits* fits, const psMetadata *header, const psArray* table,
                      const char *extname);
```

Accepts a `psArray` of `psMetadata` and writes it to the end of the file with the given `extname`.

```
bool psFitsUpdateTable(psFits* fits, const psMetadata* data, int row);
```

Writes the `psMetadata` data to a FITS table at the specified row in the current HDU. If the current HDU is not a table type, this will fail and return FALSE.

```
bool psFitsInsertTable(psFits *fits, const psMetadata *header, const psArray* table,
                       const char *extname, bool after);
```

`psFitsInsertTable` behaves in the same manner as `psFitsWriteTable`, except that the extension is inserted according to the value of the boolean `after`. If `after` is `true`, then the extension is inserted after the current `psFits` pointer; otherwise the extension is inserted before the current `psFits` pointer. **The user should beware that this is potentially a very expensive operation in terms of time, since the entire file following the inserted extension must be rewritten.**

6 Data manipulation

We require a variety of basic data manipulation functions which will act upon data (in particular, arrays/vectors). We require the following capabilities:

- Vector and image arithmetic;
- Sorting;
- Statistics;
- Matrix operations and linear algebra;
- (Fast) Fourier Transforms;
- General mathematical functions; and
- Minimization and fitting routines.

6.1 Sorting

We require the ability to sort a vector. The following function returns the vector, sorted from the smallest (i.e. most negative) value in the first element, and the largest (i.e. most positive) value in the last element. The input vector, `in`, may be sorted in-place if it is also specified as the `out` vector. This function is specified for input types `psS8, psU16, psF32, psF64`. The input and output vectors must have the same type.

```
psVector *psVectorSort(psVector *outVector, const psVector *inVector);
```

We also require the ability to sort one vector based on another. For example, we may want to sort both `x` and `y` by the value in `x`. In order to facilitate this, we will have a sort function return a vector containing the indices for the unsorted list in the order appropriate for the sorted vector, sorted from the smallest (i.e. most negative) value in the first element, and the largest (i.e. most positive) value in the last element. The output vector must be of type `psU32`. This function is specified for input types `psS8, psU16, psF32, psF64`.

```
psVector *psVectorSortIndex(psVector *outVector, const psVector *inVector);
```

The sorted vectors may be accessed in the following manner:

```
indexVector = psVectorSortIndex(NULL, x);
for (int i = 0; i < indexVector.n; i++) {
    doMyFunc(x[indexVector.arr.arr_U32[i]], y[indexVector[i].arr.arr_U32]);
}
```

6.2 Vector Operations

```
psVector *psVectorSmooth(psVector *output, const psVector *input,
                         double sigma, double Nsigma);
```

`psVectorSmooth` shall apply Gaussian smoothing to the `input` vector, with the result in the `output` vector (which shall be allocated and returned if `NULL`). The Gaussian shall be specified by a standard deviation, `sigma`, and extend `Nsigma` standard deviations.

6.3 Masks

Several functions — especially the statistical and image functions — use a mask vector and a mask value to specify values in an input list that should be excluded from the calculations. We define a mask type, which we will initially set to U8. In the event that our masks exceed eight bits, we can then change the mask definition without major changes throughout the code.

```
typedef psU8 psMaskType;
```

6.4 Statistical Functions

6.4.1 Statistical measures

We require a very general statistics function, which, given a vector of floating-point values, will be able to calculate the following population statistics:

- Minimum value;
- Maximum value;
- Sample mean;
- Sample median;
- Sample standard deviation;
- Sample upper and lower quartiles;
- Clipped mean and number of values used to calculate;
- Clipped standard deviation;
- Robust median and number of values used to calculate;
- Robust standard deviation;
- Robust upper and lower quartiles;
- Fitted mean and number of values used to calculate;
- Fitted standard deviation;

For definitions of each of these, see the accompanying Algorithms Definition Document (ADD), but in general, “sample” refers to the entire vector, “clipped” refers to clipping the distribution, “robust” refers to determining the quantities from the cumulative histogram, and “fitted” refers to fitting the data histogram with a Gaussian model. Each of these must be available from a single function:

```
psStats *psVectorStats(psStats *stats,
                       const psVector *in,
                       const psVector *errors,
                       const psVector *mask,
                       psMaskType maskVal);
```

This function takes the input data in `in` (with optional `errors` in these values; and with optional masking in `mask`, so that the user may explicitly reject specific entries) and a `psStats` structure, which will be altered and returned. The `psStats` structure includes several fields which are used for both input and output: `nSubsample` (default value of 100,000) specifies the maximum number of data points to be used for the statistics calculation. If more points than this are available, then the input vector must be randomly sampled to provide `nSubsample` measurements. `min` and `max` may be used to specify a value range for which the statistics are calculated. `binsize` specifies a choice for the robust statistics histogram bin size. If these are to be used, the user must set the corresponding `options` bits `PS_STAT_USE_RANGE` or `PS_STAT_USE_BINSIZE`. `clipSigma` specifies the number of standard deviations for which data should be clipped. `clipIter` specifies the number of iterations which should be used for clipping. The defaults for these two numbers is both 3. Since the sample statistics scale like $N \log N$, for large numbers of input data points, it is faster to use the robust statistics. Default input field values must be set by the `psStats` constructor. The input vector may be of type `psS8`, `psU16`, `psF32`, `psF64`. The `errors` must be of the same type as the `in` vector. If `errors` is not `NULL`, the calculation of certain statistics are modified: The sample mean is calculated using the formula for the weighted mean; the standard deviation is modified as specified in the ADD; the clipping used for clipped statistics are modified according to the ADD; the robust median and quartiles are modified as specified in the ADD. The mask must be of type `psMaskType`.

The `psStats` structure is defined with entries for each of the desired statistical quantities:

```
typedef struct {
    double sampleMean;                  ///< formal mean of sample
    double sampleMedian;                ///< formal median of sample
    double sampleStdev;                 ///< standard deviation of sample
    double sampleUQ;                    ///< upper quartile of sample
    double sampleLQ;                    ///< lower quartile of sample
    double robustMedian;                ///< robust median of data
    double robustStdev;                 ///< robust standard deviation of data
    double robustUQ;                    ///< robust upper quartile
    double robustLQ;                    ///< robust lower quartile
    long   robustN50;                   ///< Number of points UQ-LQ
    double fittedMean;                  ///< robust mean of data
    double fittedStdev;                 ///< robust standard deviation of data
    long   fittedNfit;                  ///< Number of points in Gauss. fit
    double clippedMean;                 ///< Nsigma clipped mean
    double clippedStdev;                ///< standard deviation after clipping
    long   clippedNvalues;              ///< number of data points used for clipped mean
    double clipSigma;                   ///< Nsigma used for clipping; user input
    int    clipIter;                    ///< Number of clipping iterations; user input
    double min;                         ///< minimum data value in data; input range
    double max;                         ///< maximum data value in data; input range
    double binsize;                     ///< binsize for robust fit (input/output)
    long   nSubsample;                  ///< maximum number of measuremenst (input)
    psStatsOptions options;             ///< bitmask of calculated values
} psStats;
```

where `psStatsOptions` is defined with entries to turn on the calculation of each of the statistics:

```
typedef enum {
    PS_STAT_SAMPLE_MEAN           = 0x000001,
    PS_STAT_SAMPLE_MEDIAN         = 0x000002,
    PS_STAT_SAMPLE_STDEV          = 0x000004,
    PS_STAT_SAMPLE_QUARTILE       = 0x000008,
    PS_STAT_ROBUST_MEDIAN         = 0x000010,
    PS_STAT_ROBUST_STDEV          = 0x000020,
```

```
    PS_STAT_ROBUST_QUARTILE       = 0x000040,
    PS_STAT_FITTED_MEAN           = 0x000080,
    PS_STAT_FITTED_STDEV          = 0x000100,
    PS_STAT_CLIPPED_MEAN          = 0x000200,
    PS_STAT_CLIPPED_STDEV         = 0x000400,
    PS_STAT_MAX                   = 0x000800,
    PS_STAT_MIN                   = 0x001000,
    PS_STAT_USE_RANGE             = 0x002000,
    PS_STAT_USE_BINSIZE           = 0x004000
} psStatsOptions;
```

A constructor is also required:

```
psStats *psStatsAlloc(psStatsOptions options);
```

```
long psVectorCountPixelMask (psVector *mask, psMaskType value);
```

This function returns the number of pixels in the `vector` which satisfy any of the mask bits. An error (eg, invalid vector) results in a return value of -1. The vector must be of the same type as psMaskValue.

6.4.2 Histograms

We also require to be able to generate histograms, given a list of upper and lower bounds for each of the bins. We define the following data structure to represent a histogram:

```
typedef struct {
    const psVector *bounds;             ///< Bounds for the bins
    psVector *nums;                     ///< Number in each of the bins
    int minNum;                         ///< Number below minimum
    int maxNum;                         ///< Number above maximum
    bool uniform;                       ///< Is it a uniform distribution?
} psHistogram;
```

In this structure, the vector `bounds` specifies the boundaries of the histogram bins, and must of type `psF32`, while `nums` specifies the number of entries in the bin, and must be of type `psF32` in order to accomodate errors. The value of `bounds.n` must therefore be 1 greater than than `nums.n`. The two values `minNum` and `maxNum` are the number of data values which fell below the lower limit bound or above the upper limit bound, respectively.

The constructors and destructor follow. We specify two constructors, so that the bounds of the bins may either be specified explicitly, or implicitly through simply specifying an upper and lower limit along with the size of the bins.

```
psHistogram *psHistogramAlloc(float lower, float upper, int n);
```

where `lower` specifies the lower bound of the histogram range, `upper` specified the upper bound of the histogram range, and `n` is the number of bins desired across the range. This constructor sets the value of `uniform` to be true.

A histogram with a more flexible bin set may be constructed with the following constructor:

```
psHistogram *psHistogramAllocGeneric(const psVector *bounds);
```

where the `psVector *bounds` (of type `psF32`) is defined by the user to specify the boundaries of the histogram bins. This constructor sets the value of `uniform` to be false.

The following function populates the histogram bins from the specified vector (`values`), and optionally the `errors` in the input values. It alters and returns the histogram `out` structure.

```
psHistogram *psVectorHistogram(psHistogram *out,
                               const psVector *values,
                               const psVector *errors,
                               const psVector *mask,
                               psMaskType maskVal);
```

The `values` vector may be of types `psU8, psU16, psF32, psF64`, with the `errors` vector of the corresponding type.

6.5 Analytical functions

6.5.1 Polynomials

PSLib provides APIs to represent and interact with polynomials in up to four dimensions, with both floating-point and double-precision numbers. In Pan-STARRS processing, the astrometry requirements push the need for at least four dimensions (x,y, color and magnitude) and double-precision (for milli-arcsec precision) versions. We must also be able to calculate the errors in the fit coefficients, as well as be able to turn on and off each coefficient.

In addition to general polynomials ($\sum_{i=0}^{n} a_i x^i$), we also use Chebyshev polynomials ($\sum_{i=0}^{n} a_i T_i(x)$), which have properties which are useful in the modeling of data over a defined domain. Note that Chebyshev polynomials should only have inputs in the range $-1 \leq x \leq +1$ (because they are bounded over this range), but we will not enforce this.

This leads us to define the following polynomial types:

```
/** One-dimensional polynomial */
typedef struct {
    psPolynomialType type;              ///< Polynomial type
    unsigned int nX;                    ///< Order Number
    psF64 *coeff;                       ///< Coefficients
    psF64 *coeffErr;                    ///< Error in coefficients
    psMaskType *mask;                   ///< Coefficient mask
} psPolynomial1D;


/** Two-dimensional polynomial */
typedef struct {
    psPolynomialType type;              ///< Polynomial type
    unsigned int nX;                    ///< Order Number in X dimension
    unsigned int nY;                    ///< Order Number in Y dimension
    psF64 **coeff;                      ///< Coefficients
    psF64 **coeffErr;                   ///< Error in coefficients
    psMaskType **mask;                  ///< Coefficients mask
} psPolynomial2D;
```

etc., up to four dimensions. `psPolynomialType` is an enumerated type specifying the type of the polynomial: ordinary or Chebyshev:

```
typedef enum {
    PS_POLYNOMIAL_ORD,                ///< Ordinary polynomial
    PS_POLYNOMIAL_CHEB                ///< Chebyshev polynomial
} psPolynomialType;
```

The constructors are:

```
psPolynomial1D *psPolynomial1DAlloc(psPolynomialType type, unsigned int nX);
psPolynomial2D *psPolynomial2DAlloc psPolynomialType type, unsigned int nX,
                                    unsigned int nY);
psPolynomial3D *psPolynomial3DAlloc(psPolynomialType type, unsigned int nX,
                                    unsigned int nY, unsigned int nZ);
psPolynomial4D *psPolynomial4DAlloc(psPolynomialType type, unsigned int nX,
                                    unsigned int nY, unsigned int nZ,
                                    unsigned int nT);
```

where `nX`, `nY`, etc specify the polynomial order in the given dimension. The coefficients, errors and masks are set initially to zero.

To evaluate the polynomials at specific coordinates, we define:

```
psF64 psPolynomial1DEval(const psPolynomial1D *poly,
                         psF64 x);
psF64 psPolynomial2DEval(const psPolynomial2D *poly,
                         psF64 x,
                         psF64 y);
psF64 psPolynomial3DEval(const psPolynomial3D *poly,
                         psF64 x,
                         psF64 y,
                         psF64 z);
psF64 psPolynomial4DEval(const psPolynomial4D *poly,
                         psF64 x,
                         psF64 y,
                         psF64 z,
                         psF64 t);
```

In the event that several evaluations are required, we also define:

```
psVector *psPolynomial1DEvalVector(const psPolynomial1D *poly,
                                   const psVector *x);
psVector *psPolynomial2DEvalVector(const psPolynomial2D *poly,
                                   const psVector *x,
                                   const psVector *y);
psVector *psPolynomial3DEvalVector(const psPolynomial3D *poly,
                                   const psVector *x,
                                   const psVector *y,
                                   const psVector *z);
psVector *psPolynomial4DEvalVector(const psPolynomial4D *poly,
                                   const psVector *x,
                                   const psVector *y,
                                   const psVector *z,
                                   const psVector *t);
```

The function shall accept input vectors of any type, converting to `psF64` as needed. In the event that the `x` and `y` vectors are of differing sizes, the function shall generate a warning, truncate the longer vector to the size of the shorter, and continuing. The precision of the output `psVector` shall be `psF64`. In evaluation, those coefficients that have the corresponding `mask` element non-zero shall not be evaluated.

6.5.2 Splines

A spline is a popular choice for fitting 1D data, such as overscans, but we neglected to define them for PSLib. We now define one-dimensional cubic splines, `psSpline1D`, which shall be incorporated into PSLib:

```
typedef struct {
    unsigned int n;                     ///< Number of spline pieces
    psPolynomial1D **spline;            ///< Array of n pointers to splines
    psVector *knots;                    ///< The boundaries between each spline piece.  Size is n+1.
    psF32 *p_psDeriv2;                  ///< Private
} psSpline1D;
```

The `psSpline1D` structure consists of an array of `n` polynomials, which are the spline pieces. Note that this means that the spline pieces may, in general, be of any order. **For the present, we shall restrict the order of the polynomials to either 1 (linear) or 3 (cubic).** All the spline pieces shall have the same order polynomial (the type of polynomial is left to the implementation). The `knots` member specifies the boundaries between each spline piece (including the two ends). The `knots` vector may be of type U32 or F32.

Of course, we require the appropriate constructors and destructor:

```
psSpline1D *psSpline1DAlloc();
```

`psSpline1DAlloc` shall allocate and return a `psSpline1D`. Since the number of spline pieces and locations of the knots depends on the input data, we do not set those here, but leave them to be set by the fitting function.

`psSpline1DAllocGeneric` shall allocate and return a `psSpline1D`, using the `bounds` to define the number of spline pieces and the `knots`. The spline pieces shall be of the specified `order`.

Also, as for the polynomials, we require evaluators. Given a `spline` and ordinate at which to evaluate, `x`, `psSpline1DEval` shall evaluate and return the value of the spline at the ordinate. If the ordinate is outside the bounds, then the function shall generate a warning, and extrapolate the spline to the ordinate and return the value. Similarly, `psSpline1DEvalVector` shall return a vector of evaluated values for an input vector of ordinates.

```
float psSpline1DEval(const psSpline1D *spline, float x);
psVector *psSpline1DEvalVector(const psSpline1D *spline, const psVector *x);
```

6.5.3 Gaussians

Gaussians are used extensively in any data-analysis system, in particular to represent a real population distribution. We require a function to evaluate a Gaussian for a given coordinate.

The Gaussian evaluation is provide by:

```
float psGaussian(float x, float mean, float sigma, bool normal);
```

which evaluates a Gaussian with the given `mean` and `sigma` at the given coordinate `x`. If `normal` is true, the Gaussian is normalized (total integral = 1), otherwise, the Gaussian is non-normalized (central peak value = 1). The evaluated Gaussian is:

$$\frac{1}{\sqrt{2\pi\sigma^2}} exp^{-\frac{(x-mean)^2}{2\sigma^2}}$$

In the case of the non-normalized Gaussian, the leading coefficient is dropped.

6.6 Minimization and fitting routines

We require a general minimization routine, a routine that will specifically minimize χ^2 given a list of data with associated errors, and functions that will analytically determine the best polynomial and spline fits by minimizing χ^2.

We specify two minimization engines, Levenberg-Marquardt and Powell, since the former is relatively robust, but requires derivatives to be known, while the latter does not require derivatives to be known, but since it needs to calculate the derivatives on the fly, is more computationally intensive.

We define the structure `psMinimization` to carry in user-defined limits on the minimization process, and to carry out information about the minimization analysis. The maximum number of iterations is specified by `maxIter`, while the maximum tolerance for convergence is `tol`. The output information carried by the structure consists of the value of the function at the minimum (`value`), the number of iterations performed (`iter`) and last change in tolerance before returning (`lastDelta`).

```
typedef struct {
    const int maxIter;                  ///< Maximum number of iterations
    const float tol;                    ///< Tolerance to reach
    float value;                        ///< Value after minimization
    int iter;                           ///< Actual number of iterations performed
    float lastDelta;                    ///< Last change before quitting
} psMinimization;
```

We define the `psMinConstrain` structure to define values which constrain the allowed parameter values. The `paramMask` vector defines the free (0) or frozen (not 0) parameters. The `paramMin` defines the minimum allowed value for each parameter, while `paramMax` defines the maximum allowed value for each parameter. During the analysis, parameters which would trend beyond these limits saturate to the limit. The `paramDelta` specifies the maximum allowed absolute value of the swing of a given parameter. If the delta for a specific parameter would be larger than this, the swing is saturated to the limit (with the correct sign). Any of these parameter vectors may be set to `NULL`, in which case the concept is ignored in the analysis.

```
typedef struct {
    psVector *paramMask;                ///< valid / invalid parameters
    psVector *paramMax;                 ///< max allowed parameters
    psVector *paramMin;                 ///< min allowed parameters
    psVector *paramDelta;               ///< max allowed param swing
} psMinConstrain;
```

The corresponding allocators are:

```
psMinimization *psMinimizationAlloc(int maxIter, float tol);
psMinConstrain *psMinConstrainAlloc();
```

and the parameter vectors are initially set to `NULL`.

6.6.1 Levenberg-Marquardt

Consider a function of a collection of parameters, `params`, which is evaluated at a position, `x`, which returns a single floating point value which is the value of the function given the parameters and coordinate vectors, along with the derivatives of the function with respect to each of the parameters, `deriv`:

```
typedef float (*psMinimizeLMChi2Func)(psVector *deriv, const psVector *params, const psVector *x);
```

Then `psMinimizeLMChi2` shall fit the specified function, `func`, to a set of measurements, `x,y,yWt`, using the Levenberg-Marquardt method:

```
bool psMinimizeLMChi2(psMinimization *min,
                      psImage *covar,
                      psVector *params,
                      psMinConstrain *constrain,
                      const psArray  *x,
                      const psVector *y,
                      const psVector *yWt,
                      psMinimizeLMChi2Func func);
```

The function shall return `false` in the event that there was an error (bad input, too many iterations), and also call `psError`. It is not an error for the minimization to reach one of the parameter limits.

The minimization specification, `min`, shall be modified with the `value`, `iter` and `lastDelta` members updated with the values appropriate from the minimization.

On calling the minimizer, the `params` vector shall consist of a "best guess" for the parameters that minimize the model function, `func`. On successful completion, the input parameters, `params`, shall be updated with the values that minimize the input function. The function shall also update the covariance matrix, `covar`, in the event that it is non-`NULL`. The function shall generate in error in the event that `covar` is non-`NULL` and is not a square matrix with size matching that of `params`.

Parameters that have a corresponding `paramMask` entry that is non-zero are to be held fixed by the minimizer. It shall be an error for `paramMask` not to be of the same dimension as `params`.

The measurement ordinates, `x`, shall consist of multiple vectors, each of which may be passed to the model `func`. If the measurement coordinates, `y`, and weights, `yWt`, are not of the same length as the ordinates array, `x`, then the function shall generate a warning, and truncate the longest of the array/vectors to match the length of the shortest. The vectors contained within the `x` array, and the `y` and `yWt` vectors must be of type `psF64`. The `yWt` vector may be `NULL`, in which case the errors shall be assumed to be identical.

`paramMask` must be of type `psMaskType`, while `params` must be of type `psF64`. The `func` function must be valid only for type `psF64`.

```
bool psMinimizeGaussNewtonDelta(psVector *delta,
                                const psVector *params,
                                const psVector *paramMask,
                                const psArray  *x,
                                const psVector *y,
                                const psVector *yErr,
                                psMinimizeLMChi2Func func);
```

The function `psMinimizeGaussNewtonDelta` can be used to evaluate the goodness of a fit. This function returns the distances of the current parameter set from the minimization parameters expected from Gauss-Newton minimization. The arguments have the same meaning as for `psMinimizeLMChi2`. The returned vector, `delta`, consists of the distances. This vector must be pre-allocated to the dimenstions of `params`.

6.6.2 Powell

As for the LM minimizer, consider a function of a collection of parameters, `params`, and (possibly several) coordinate vectors (which we represent as an array of vectors), `coord`, and returns a single floating point value which is the value of the function given the parameters and coordinate vectors, but now the derivatives are not known:

```
typedef float (*psMinimizePowellFunc)(const psVector *params, const psArray *coords);
```

Then `psMinimizePowell` shall minimize the specified function, `func`, using the Powell method:

```
bool psMinimizePowell(psMinimization *min,
                      psVector *params,
                      const psVector *paramMask,
                      const psArray *coords,
                      psMinimizePowellFunc func);
```

The inputs and general behavior of this function is the same as for `psMinimizeLM`, except for the absence of the covariance matrix, `covar` **why does this not have a covar matrix?**

6.6.2.1 Minimizing χ^2 with Powell

We require a front-end to the Powell minimizer to allow fitting general functions to data.

```
typedef psVector* (*psMinimizeChi2PowellFunc)(const psVector *params, const psArray *coords);

bool psMinimizeChi2Powell(psMinimization *min,
                          psVector *params,
                          psMinConstrain *constrain,
                          const psArray *coords,
                          const psVector *value,
                          const psVector *error,
                          psMinimizeChi2PowellFunc model);
```

The inputs and general behavior of `psMinimizeChi2Powell` is similar as for `psMinimizePowell`, with the exception that instead of being provided the function to be minimized, it is provided a model to fit and must calculate the χ^2 to be minimized itself.

why does this not have a covar matrix?

unify the param names with psMinimizeLMChi2Func?

For this purpose, `value` shall contain measured values at the coordinates, and `error` may either be non-`NULL`, in which case it contains the errors in the measured values; otherwise the errors shall assumed to be unity for the purpose of fitting where the errors are all identical (and possibly unknown).

Furthermore, the `model` function provided by the user shall return a vector of values (instead of a single value, as was the case for `psMinimizePowell`), corresponding to the model predictions to be compared against the measured values. If the lengths of the `value`, `error` (if provided) vectors and the vector returned by the `model` function differ, then `psMinimizeChi2Powell` shall generate a warning, truncate the vectors to the length of the shortest and proceed (only one error should be generated per call).

6.6.3 Analytical fits

```
psPolynomial1D *psVectorFitPolynomial1D(psPolynomial1D *poly,
                                        const psVector *mask,
                                        psMaskType      maskValue,
                                        const psVector *f,
                                        const psVector *fErr,
                                        const psVector *x);
psPolynomial2D *psVectorFitPolynomial2D(psPolynomial2D *poly,
                                        const psVector *mask,
                                        psMaskType      maskValue,
                                        const psVector *f,
                                        const psVector *fErr,
                                        const psVector *x,
                                        const psVector *y);
psPolynomial3D *psVectorFitPolynomial3D(psPolynomial3D *poly,
                                        const psVector *mask,
                                        psMaskType      maskValue,
                                        const psVector *f,
                                        const psVector *fErr,
                                        const psVector *x,
                                        const psVector *y,
                                        const psVector *z);
psPolynomial4D *psVectorFitPolynomial4D(psPolynomial4D *poly,
                                        const psVector *mask,
                                        psMaskType      maskValue,
                                        const psVector *f,
                                        const psVector *fErr,
                                        const psVector *x,
                                        const psVector *y,
                                        const psVector *z,
                                        const psVector *t);
```

These functions return the polynomial that best fits the input data. The provided polynomial, `poly`, specifies the fit order, and will be returnd with the best-fit coefficients. The dependent variable and its error (`f, fErr`) are provided along with the appropriate number of independent variables (`x`, `y`, etc). In the special case of 1D fitting, the independent variable list, `x` may be `NULL`, in which case the vector index is used. The dependent variable error, `yErr` may be null, in which case the solution is determined in the assumption that all data errors are equal. This function must be valid only for input data types of `psF32`, `psF64`. The `mask` and `maskValue` entries may be used to specify an input data set to ignore. All of the input vectors must be the same length. Coefficients in the `poly` that are masked shall not be fit.

```
psPolynomial1D *psVectorClipFitPolynomial1D(psPolynomial1D *poly,
                                            psStats        *stats,
                                            const psVector *mask,
                                            psMaskType      maskValue,
                                            const psVector *f,
                                            const psVector *fErr,
```

```
                                        const psVector     *x);
psPolynomial2D *psVectorClipFitPolynomial2D(psPolynomial2D *poly,
                                        psStats            *stats,
                                        const psVector     *mask,
                                        psMaskType          maskValue,
                                        const psVector     *f,
                                        const psVector     *fErr,
                                        const psVector     *x,
                                        const psVector     *y);
psPolynomial3D *psVectorClipFitPolynomial3D(psPolynomial3D *poly,
                                        psStats            *stats,
                                        const psVector     *mask,
                                        psMaskType          maskValue,
                                        const psVector     *f,
                                        const psVector     *fErr,
                                        const psVector     *x,
                                        const psVector     *y,
                                        const psVector     *z);
psPolynomial4D *psVectorClipFitPolynomial4D(psPolynomial4D *poly,
                                        psStats            *stats,
                                        const psVector     *mask,
                                        psMaskType          maskValue,
                                        const psVector     *f,
                                        const psVector     *fErr,
                                        const psVector     *x,
                                        const psVector     *y,
                                        const psVector     *z,
                                        const psVector     *t);
```

These functions replicate the capability of the vector fitting functions, but also include an iterative clipping stage. The clipping parameters are defined by the `stats` entry, to which are also returned the clipped statistics for the final residuals. Thus, if N clipping iterations are requested, the function performs the fit, constructs the residuals, measures the residual scatter, and masks the outlier vector elements. This cycle is repeated N times, though on the last iteration, no additional masking is performed. The provided mask must be respected, and any additionally masked elements are also masked with the same mask vector, which must be provided. The elements masked by this routine are given the mask value of 1 and may thus be distinguished from input masked elements if they use a different mask value.

```
psSpline1D *psVectorFitSpline1D(const psVector *x, const psVector *y);
```

`psVectorFitSpline1D` shall return the spline that best fits the given combination of ordinates (`x`) and coordinates (`y`). As is the case for `psVectorFitPolynomial1D`, if `x` is `NULL`, then the index of `y` shall be used as the ordinate. This function must be valid only for types `psF32`, `psF64`.

6.6.4 Additional polynomial functions

EAM to explain in more detail (TBD)

```
psPolynomial4D *psVectorChiClipFitPolynomial4D(
    psPolynomial4D *poly,
    psStats *stats,
    const psVector *mask,
    psMaskType maskValue,
```

```
    const psVector *f,
    const psVector *fErr,
    const psVector *x,
    const psVector *y,
    const psVector *z,
    const psVector *t);
```

psVectorChiClipFitPolynomial4D shall perform a vector clip-fit of a polynomial to the input data (f,fErr,x,y,z,t), based on significance of deviations

6.7 Image Operations

We require a variety of functions to manipulate these image structures, including creation, destruction, input, output, and various manipulations of the pixels. The required functions are listed below, and fall into several categories.

6.7.1 Image Structure Manipulation

```
psImage *psImageSubset(psImage *image, psRegion region);
```

Define a subimage of the specified area of the given image. This function must raise an error if the requested subset area lies outside of the parent image and return NULL. The argument image is the parent image, region.x0, region.y0 specify the starting pixel of the subraster, and region.x1,region.y1 specify the extent of the desired subraster. Note that the row and column of this "upper right-hand corner" are *NOT* included in the region. Note that, if the image is itself a subimage, then the region coordinates correspond to coordinates of the parent image of image. The only exception to this is in the event that x1 or y1 are non-positive, in which case they shall be interpreted as being relative to the upper-bounds of image in that dimension. The entire resulting subraster will be contained within the raster of the input image; if the requested bounds of region fall beyond the bounds of the input image, they should saturate on the input image region (in analogy with the behavior of psRegionForImage). Note that the refCounter for the parent should be incremented. This function must be defined for the following types: psU8, psU16, psS8, psS16, psF32, psF64, psC32, psC64.

```
psImage *psImageCopy(psImage *output, const psImage *input, psElemType type);
```

Create a copy of the specified image, converting the type in the process. If the output target pointer is not NULL, place the result in the specified structure. The output image data must be allocated as a single, contiguous block of memory. The output image may not be the input image. The col0,row0 of the input image shall be preserved. This function must be defined for the following types: psU8, psU16, psS8, psS16, psF32, psF64, psC32, psC64.

```
psImage *psImageTrim(psImage *image, psRegion region);
```

Trim the specified image in-place, which involves shuffling the pixels around in memory. The region to be kept is defined by the lower-left corner, region.x0,region.y0, and the upper-right corner, region.x1,region.y1. Note that the row and column of the "upper right-hand corner" are *NOT* included in the region. Note that, if the image is itself a subimage, then the region coordinates correspond to coordinates of the parent image of image. However, in the event that region.x1 or region.y1 are negative, they shall be interpreted as being relative to the upper bounds of the input image in that dimension.

The function shall generate an error if the specified region is outside the bounds of the input image. Any children of the input image shall be freed by `psImageTrim` before the trim takes place.

The following function flips the given image in the x and/or y direction. Note that a request for both an x and a y flip is identical to a 180° rotation.

```
psImage *psImageFlip(psImage *output, const psImage *input,
                     bool xFlip, bool yFlip);
```

6.7.2 Image Pixel Extractions

```
typedef enum {
    PS_CUT_X_POS,                    ///< Cut in positive x direction
    PS_CUT_X_NEG,                    ///< Cut in negative x direction
    PS_CUT_Y_POS,                    ///< Cut in positive y direction
    PS_CUT_Y_NEG                     ///< Cut in negative y direction
} psImageCutDirection;

psVector *psImageRow(psVector *out,
                     const psImage *input,
                     int row);
psVector *psImageCol(psVector *out,
                     const psImage *input,
                     int column);
```

These two functions extract a single complete `row` or `column` from the `image` and return it to the provided `vector`, allocating it if it is `NULL`. If the input image is a subimage, the `row` and `column` arguments refer to parent coordinates. These are provided as fast, simple implementations of data extraction. For more sophisticated operations, the following two functions should be used.

```
psVector *psImageSlice(psVector *out,
                       psPixels *coords,
                       const psImage *input,
                       const psImage *mask,
                       psMaskType maskVal,
                       psRegion region,
                       psImageCutDirection direction,
                       const psStats *stats);
```

Extract pixels from a rectilinear region to a vector (array of floats). The output vector contains either `region.x1-region.x0` or `region.y1-region.y0` elements, based on the value of the direction: e.g., if `direction` is `PS_CUT_X_POS`, there are `region.x1-region.x0` elements. The region to be "sliced" is defined by the lower-left corner, `region.x0,region.y0`, and the upper-right corner, `region.x1,region.y1`. Note that the row and column of the "upper right-hand corner" are *NOT* included in the region. In the event that `region.x1` or `region.y1` are negative, they shall be interpreted as being relative to the size of the input image in that dimension. If the input image is a subimage, the `region` argument refers to parent coordinates.

The pixel region defined by the coordinate pair `start` & `stop` is collapsed in the direction perpendicular to that specified by `direction`, and each element of the output vectors is derived from the statistics of the pixels at that direction coordinate. The statistic used to derive the output vector value is specified by `stats`. If `mask` is non-`NULL`, pixels for which the corresponding `mask` pixel matches `maskVal` are excluded from operations. If `coords` is not `NULL`,

the calculated coordinates along the slice are returned in this array of pixels. Only one of the statistics choices may be specified, otherwise the function must return an error. This function must be defined for the following types: psS8, psU16, psF32, psF64.

```
psVector *psImageCut(psVector *out,
                     psVector *cutCols,
                     psVector *cutRows,
                     const psImage *input,
                     const psImage *mask,
                     psMaskType maskVal,
                     psPlane *start,
                     psPlane *stop,
                     unsigned int nSamples,
                     psImageInterpolateMode mode);
```

Extract pixels along a line segment, (region.x0,region.y0) to (region.x1,region.y1), on the image to a vector of the same data type. The line segment is sampled nSamples times, hence the output vector contains nSamples elements. If the input image is a subimage, the region argument refers to parent coordinates. The interpolation method used to derive the output vector value at each sample along the line segment is specified by mode. If the mask is non-NULL, then pixels for which the corresponding mask value is maskVal are not included in the interpolation. This function must be defined for the following types: psS8, psU16, psF32, psF64.

```
psVector *psImageRadialCut(psVector *out,
                           const psImage *input,
                           const psImage *mask,
                           psMaskType maskVal,
                           float x,
                           float y,
                           const psVector *radii,
                           const psStats *stats);
```

Extract radial region data to a vector. A vector is constructed where each vector elements is derived from the statistics of the pixels which land within one of a sequence of radii. The radii are centered on the image pixel coordinate x,y, and are defined by the sequence of values in the vector radii. If the input image is a subimage, the x,y arguments refer to parent coordinates. The specific algorithm which must be used is described in the PSLib ADD (PSDC-430-006). The statistic used to derive the output vector value is specified by stats. Only one of the statistics choices may be specified, otherwise the function must return an error. If mask is non-NULL, pixels for which the corresponding mask pixel matches maskVal are excluded from operations. This function must be defined for the following types: psS8, psU16, psF32, psF64.

6.7.3 Image Geometry Manipulation

Several functions which manipulate the image geometry require the specification of the interpolation scheme to be used. This information is carried by the following enum:

```
typedef enum {
    PS_INTERPOLATE_FLAT,
    PS_INTERPOLATE_BILINEAR,
    PS_INTERPOLATE_LANCZOS2,
    PS_INTERPOLATE_LANCZOS3,
```

```
    PS_INTERPOLATE_LANCZOS4,
    PS_INTERPOLATE_BILINEAR_VARIANCE,
    PS_INTERPOLATE_LANCZOS2_VARIANCE,
    PS_INTERPOLATE_LANCZOS3_VARIANCE,
    PS_INTERPOLATE_LANCZOS4_VARIANCE
} psImageInterpolateMode;
```

The first five are fairly straightforward. The last four, however, require some explanation. When attempting to account for noise in a CCD image, it is customary to carry an additional image containing the variance for each pixel. When operating on the CCD image (performing some transformation), we want to perform the same operation on the variance image, but the weights of the different input pixels contributing to the output pixel must be squared in order to propagate the noise (adding in quadrature).

```
psImage *psImageRebin(psImage *out, const psImage *in,
                      const psImage *mask,
                      psMaskType maskVal,
                      int scale, const psStats *stats);
```

Rebin image to new scale. A new image is constructed in which the dimensions are reduced by a factor of `1 / scale`. The `scale`, always a positive number, is equal in each dimension and specified the number of pixels used to define a new pixel in the output image. The output image is generated from all input image pixels. Care must be taken on the image boundary if the image dimensions are not divisible by the scaling factor. In those regions, the output pixel must be constructed from the available input pixels. Each pixel in the output image is derived from the statistics of the corresponding set of input image pixels based on the statistics specified by `stats`. Only one of the statistics choices may be specified, otherwise the function must return an error. If `mask` is non-`NULL`, pixels for which the corresponding `mask` pixel matches `maskVal` are excluded from operations. This function must be defined for the following types: `psS8`, `psU16`, `psF32` and `psF64`.

```
psImage *psImageResample(psImage *out, const psImage *in,
                         int scale, psImageInterpolateMode mode);
```

Resample image to new scale. A new image is constructed in which the dimensions are increased by a factor of `scale`. The `scale`, always a positive number, is equal in each dimension. The output image is generated from all input image pixels. Each pixel in the output image is derived by interpolating between neighboring pixels using the specified interpolation method (`mode`).

```
psImage *psImageRotate(psImage *out, const psImage *input, float angle,
                       double complex exposed, psImageInterpolateMode mode);
```

Rotate the input image by given angle, specified in radians. The output image must contain all of the pixels from the input image in their new frame. Pixels in the output image which do not map to input pixels should be set to `exposed`, cast to the same type as the image. The center of rotation is always the center pixel of the image. The rotation is specified in the sense that a positive angle is an anti-clockwise rotation. This function must be defined for the following types: `psU8`, `psU16`, `psS8`, `psS16`, `psF32`, `psF64`, `psC32`, `psC64`.

```
psImage *psImageShift(psImage *out, const psImage *input,
                      float dx, float dy, double complex exposed, psImageInterpolateMode mode);
```

Shift image by an arbitrary number of pixels (dx,dy) in either direction. If the shift values are fractional, the output pixel values should interpolate between the input pixel values. The output image has the same dimensions as the input image. Pixels which fall off the edge of the output image are lost. Newly exposed pixels are set to the value given by exposed, cast to the same type as the image. This function must be defined for the following types: psU8, psU16, psS8, psS16, psF32, psF64, psC32, psC64.

```
psImage *psImageRoll(psImage *out, const psImage *input, int dx, int dy);
```

Roll image by an integer number of pixels (dx,dy) in either direction. The output image is the same dimensions as the input image. Edge pixels wrap to the other side (no values are lost). This function must be defined for the following types: psU8, psU16, psS8, psS16, psF32, psF64, psC32, psC64.

```
psImage *psImageTransform(psImage *output,
                          psPixels **blankPixels,
                          const psImage *input,
                          const psImage *inputMask,
                          psMaskType inputMaskVal,
                          const psPlaneTransform *outToIn,
                          psRegion region,
                          const psPixels *pixels,
                          psImageInterpolateMode mode,
                          double exposedValue);
```

Transform the input image according the supplied transformation. The size of the transformed image is defined by the region (size region.x1 - region.x0 by region.y1 - region.y0, with out->x0 = region.x0 and out->y0 = region.y0). If the input image is itself a subimage, then the region refers to the parent image coordinates.

If the inputMask is non-NULL, those pixels in the inputMask matching inputMaskVal are to be ignored in the transformation. The inputMask must be of type psMaskType, and of the same size as the input, otherwise the function shall generate an error and return NULL. The transformation outToIn specifies the coordinates in the input image of a pixel in the output image — note that this is the reverse of what might be naively expected, but it is what is required in order to use psImagePixelInterpolate. If the pixels array is non-NULL, it shall consist of psPixelCoords, and only those pixels in the output image shall be transformed; otherwise, the entire image is generated. The interpolation is performed using the specified interpolation mode. Where a pixel in the output image does not correspond to a pixel in the input image (or all appropriate pixels in the input image are masked), the value shall be set to exposed, and the pixel added to the appropriate list of pixels, blankPixels (which shall be allocated if blankPixels is non-NULL and *blankPixels is NULL), for return to the user. This function must be capable of handling the following types for the input (with corresponding types for the output): psF32, psF64.

Description required for psImageUnbin (TBD)

```
psImage *psImageUnbin(psImage *out, const psImage *in, int DX, int DY, int dx, int dy);
```

Can “out” be NULL? I’m not sure this is the best way to specify this function for a library. (TBD)

6.7.4 Image Statistical Functions

```
psStats *psImageStats(psStats *stats,
                      const psImage *in,
```

```
                    const psImage *mask,
                    psMaskType maskVal);
```

Determine statistics for image (or subimage). The statistics to be determined are specified by `stats`. The `mask` allows pixels to be excluded if their corresponding mask pixel value matches the value of `maskVal`. This function must be defined for the following types: `psS8`, `psU16`, `psF32`, `psF64`.

```
psHistogram *psImageHistogram(psHistogram *out,
                              const psImage *in,
                              const psImage *mask,
                              psMaskType maskVal);
```

Construct a histogram from an image (or subimage). The histogram to generate is specified by `psHistogram hist` (see section 6.4.2). The `mask` and `maskVal` entries are passed to the psLib statistics function used to calculate the ensemble statistics. This function must be defined for the following types: `psS8`, `psU16`, `psF32`, `psF64`.

```
long psImageCountPixelMask (psImage *mask, psRegion region, psMaskType value);
```

This function returns the number of pixels in the image region which satisfy any of the mask bits. An error (eg, invalid image, invalid region) results in a return value of -1. The `region` refers to the pixels of the `mask`; if `mask` is a subimage, the region must be defined relative to the parent pixel coordinate.

```
psPolynomial2D *psImageFitPolynomial(psPolynomial2D *coeffs, const psImage *input);
```

Fit a 2-D Chebychev polynomial surface to an image. The input structure `coeffs` contains the desired order and terms of interest. This function must be defined for the following types: `psS8`, `psU16`, `psF32`, `psF64`.

```
psImage *psImageEvalPolynomial(psImage *input, const psPolynomial2D *coeffs);
```

Evaluate a 2-D polynomial surface for the image pixels. Given the input polynomial coefficients, set the image pixel values on the basis of the polynomial function. This function must be defined for the following types: `psS8`, `psU16`, `psF32`, `psF64`.

```
double complex psImagePixelInterpolate(const psImage *input, float x, float y,
                              const psImage *mask, psMaskType maskVal,
                              double complex unexposedValue, psImageInterpolateMode mode);
```

Perform interpolation of image pixel values to the given fractional coordinate `x,y`. The function returns the interpolated value of the image at the given fractional pixel coordinates, based on the specified interpolation `mode`. The `mask` allows pixels to be excluded if their corresponding mask pixel value matches the value of `maskVal`. This function will likely be implemented as a macro for processing speed. It may also be necessary to define a setup macro which pre-calculates certain values which would be reused in a loop.

```
psStats *psImageBackground(const psImage *image, // Image for which to get the background
                           const psImage *mask, // Mask image
                           psMaskType maskValue, // Mask pixels which this mask value
```

```
                        double fmin, // Fraction to return in the lower quartile field (0.25 for LQ)
                        double fmax, // Fraction to return in the upper quartile field (0.75 for LQ)
                        long nMax,   // Maximum number of pixels to subsample
                        psRandom *rng // Random number generator (for pixel selection)
    );
```

`psImageBackground` shall measure the median background level for the input `image` from a random subsample of pixels, with the median in the `robustMedian` of the returned `psStats`. If `mask` is non-`NULL`, then pixels with a corresponding mask value with the `maskValue` set shall be excluded from the calculation. A maximum of `nMax` pixels shall be selected randomly from the image, using the provided random number generator, `rng`. The function shall also set the values for which the cumulative fractions are `fmin` and `fmax` in the `robustLQ` and `robustUQ` fields of the returned `psStats`, respectively.

6.7.5 Image Pixel Manipulations

```
int psImageClip(psImage *input, double min, double vmin, double max, double vmax);
```

Clip image values outside of range to given values. All pixels with values `< min` are set to the value `vmin`. All pixels with values `> max` are set to the value `vmax`. Returns the number of clipped pixels. This function must be defined for the following types: `psU8`, `psU16`, `psS8`, `psS16`, `psF32`, `psF64`, `psC32`, `psC64`. The arguments (`min`, `max`, etc) must be cast to the appropriate types to match the image data. If the input parameters `vmin` or `vmax` are out of bounds for the image pixel type, the function must raise an error. It is not an error for `min` or `max` to be out of range. In the case of complex numbers, the input parameters `min` and `max` must be compared against the absolute value of the pixel values. The values to which complex image data are clipped employ the provided value for the real component and 0.0 for the imaginary component.

```
int psImageClipComplexRegion(psImage *input, double complex min,
                             double complex vmin, double complex max,
                             double complex vmax);
```

Clip image values outside of range to given values. All pixels with values `< min` are set to the value `vmin`. All pixels with values `> max` are set to the value `vmax`. Returns the number of clipped pixels. This function must be defined for the following types: `psC32`, `psC64`. The arguments (`min`, `max`, etc) define a rectangular region in complex space; data values inside this regions are unchanged while those outside are set to either `vmax` (if either their real or imaginary portions are greater than the corresponding values of `max`) or `vmin` (in all other cases). If the input parameters `vmin` or `vmax` are out of bounds for the image pixel type, the function must raise an error. It is not an error for `min` or `max` to be out of range.

```
int psImageClipNaN(psImage *input, float value);
```

Clip `NaN` image pixels to given value. Pixels with `NaN`, `+Inf` or `-Inf` values are set to the specified value. Returns the number of clipped pixels. This function must be defined for the following types: `psF32`, `psF64`, `psC32`, `psC64`. In the case of complex values, if either the real or imaginary part have the value `NaN`, then that component will be set to the specified value.

```
int psImageOverlaySection(psImage *image, const psImage *overlay,
                          int x0, int y0, const char *op);
```

Overlay subregion of image with another image. Replace the pixels in the `image` which correspond to the pixels in `overlay` with values derived from the values in `image` and `overlay` based on the given operator `op`. Valid operators are = (set image value to overlay value), + (add overlay value to image value), - (subtract overlay from image), * (multiply overlay times image), / (divide image by overlay). This function must be defined for the following types: `psU8`, `psU16`, `psS8`, `psS16`, `psF32`, `psF64`, `psC32`, `psC64`. The two input images must have the same datatype and the output image must be constructed with that same datatype. The return value shall be the number of pixels overlaid.

6.7.6 Image Local Bicube

The following functions provide interpolations of image data values based on bicubic interpolation.

This function fits a 2D 2nd order polynomial to the 9 pixels centered on the coordinate x,y.

```
psPolynomial2D *psImageBicubeFit(const psImage *image, int x, int y);
```

This function detemines the min (or max) of the special 2D 2nd order polynomial representing the fit to 9 pixels of an image.

```
psPlane psImageBicubeMin(const psPolynomial2D *poly);
```

6.7.7 JPEG operations

The following functions and structures are used to convert an image in memory to a output JPEG file. These functions allow the user to define a color map (from a list of predefined color maps) and to define the dynamic range of the translation from data values to JPEG color values.

The representation of a JPEG colormap is given by the following structure:

```
typedef struct {
    psVector *red;
    psVector *green;
    psVector *blue;
} psImageJpegColormap;
```

The colormap is just a translation from a `psImage` data value to a JPEG image data value. The colormap may be used to construct either single channel or multichannel images.

The following function allocates, but does not specify, a JPEG colormap:

```
psImageJpegColormap *psImageJpegColormapAlloc();
```

The following function sets the colormap values based on the named colormap. Currently defined colormaps have the names `greyscale` (or `grayscale`), `-grayscale` (or `-greyscale`), `rainbow`, `heat`.

```
psImageJpegColormap *psImageJpegColormapSet(psImageJpegColormap *map, const char *name);
```

The following function writes out the specified image as a JPEG file using the supplied colormap. The output goes to the specified filename. This function performs a single-channel JPEG conversion (the values of the single image determine the colors).

```
bool psImageJpeg(const psImageJpegColormap *map, const psImage *image,
                 const char *filename, float min, float max);
```

6.7.8 Mask operations

```
psImage *psImageGrowMask(psImage *out, const psImage *in, psMaskType maskVal,
                         unsigned int growSize, psMaskType growVal);
```

`psImageGrowMask` grows specified values on the input mask image, `in`, returning the result. If `out` is `NULL`, then a new image of the same type and dimension as `in` shall be allocated and returned; otherwise `out` shall be modified. If `out` is non-`NULL` and does not have the same size and type as `in`, the function shall generate an error and return `NULL`. Pixels in the `in` image within `growSize` pixels (either horizontal or vertical) of a pixel which matches the `maskVal` shall have the corresponding pixel in the `out` image set to the `growValue`.

```
psImage *psPixelsToMask(psImage *out, const psPixels *pixels, psRegion region, psMaskType maskVal);
psPixels *psPixelsFromMask(psPixels *out, const psImage *mask, psMaskType maskVal);
```

`psPixelsToMask` shall return an image of type `psMaskType` with the `pixels` lying within the specified `region` set to the `maskVal`. The `out` image shall be modified if supplied, or allocated and returned if `NULL`. The size of the output image shall be `region.x1 - region.x0` by `region.y1 - region.y0`, with `out->x0 = region.x0` and `out->y0 = region.y0`. In the event that either of `pixels` or `region` are `NULL`, the function shall generate an error and return `NULL`. If `out` is not supplied, then the constructed image is not a subimage. If `out` is supplied and is a subimage, the pixels refer to the parent image coordinates.

`psMaskToPixels` shall return a `psPixels` containing the coordinates in the `mask` that match the `maskVal`. The `out` pixel list shall be modified if supplied, or allocated and returned if `NULL`. In the event that `mask` is `NULL`, the function shall generate an error and return `NULL`. If `mask` is a subimage, the pixels refer to the parent image coordinates.

```
void psImageMaskRegion(psImage *image, psRegion region, const char *op, psMaskType maskValue);
void psImageKeepRegion(psImage *image, psRegion region, const char *op, psMaskType maskValue);
```

These two complementary functions set bit specified bits (`maskValue`) in the mask `image` interior to or exterior to the specified `region`. The first function, `psImageMaskRegion`, sets the bits inside the region, ignoring pixels outside. The second, `psImageKeepRegion`, sets the bits outside the region, ignoring pixels inside. The pixel values are set by combining the existing pixel value and the given `maskValue` with a logical operation. The allowed operations are `=`, `AND`, `OR` and `XOR`. If `image` is a subimage, the region refers to the parent image coordinates.

```
void psImageMaskCircle(psImage *image, double x, double y, double radius, const char *op,
                       psMaskType maskValue);
void psImageKeepCircle(psImage *image, double x, double y, double radius, const char *op,
                       psMaskType maskValue);
```

These two complementary functions set bit specified bits (`maskValue`) in the mask `image` interior to or exterior to the specified circle, defined by the center coordinates `x,y` and a `radius`. The first function, `psImageMaskCircle`, sets the bits inside the circle, ignoring pixels outside. The second, `psImageKeepCircle`, sets the bits outside the circle, ignoring pixels inside. The pixel values are set by combining the existing pixel value and the given `maskValue` with a logical operation. The allowed operations are `=`, `AND`, `OR` and `XOR`. If `image` is a subimage, the coordinates refers to the parent image coordinates.

6.8 Vector and Image Arithmetic

We will need to be able to perform various operations on vectors and images, e.g. dividing one image by another, subtracting a vector from an image, etc. Both binary operations and unary operations are required. To avoid the burden of memorizing a plethora of APIs, we specify two generic APIs for the binary and unary operations.

```
psMathType *psBinaryOp(psPtr out, const psPtr in1, const char *op, const psPtr in2);
psMathType *psUnaryOp(psPtr out, const psPtr in, const char *op);
```

These functions determine the type of the operands on the basis of their `psMathType` elements, which always are the first elements. Note that these functions return a pointer to the appropriate type for the operation. Since the result may be cast to `psMathType`, the resulting type may be determined by examining the return value. It is expected that the implementation of these functions will employ pre-processor macros to perform the onerous task of creating the loops. An attempt to perform an arithmetic operation on an object of dimension `PS_DIMEN_OTHER` should produce an error. Operations between data structures with different types (e.g., `psS32` and `psF32`) are not allowed and must raise an error (it is the responsibility of calling functions to perform type conversions). Operations between data structures with incompatible sizes are not allowed. However, operations between data elements of different rank (scalar, vector, image) are allowed, and defined below. These functions are valid for all data types `psU8`, `psU16`, `psS8`, `psS16`, `psF32`, `psF64`, `psC32`, `psC64`.

Binary operations between an image and a vector have a potential ambiguity — do the vector elements correspond to the rows or the columns? For this reason, we define two vector types: a "vector" (`PS_DIMEN_VECTOR`), and a "transposed vector" (`PS_DIMEN_TRANSV`). We specify that a "vector", when involved in binary operations on an image, acts on all rows of the image, while a "transposed vector" in the same context acts on all columns. Vectors, when created, will be created as "vectors", but may be converted to "transposed vectors" by setting `vector->type.dimen = PS_DIMEN_TRANSV`.

It is further desirable to allow scalar values to be used within these functions. These functions may take a pointer to type `psScalar`, which is freed by the function. This allows one to write the following lines to take the sine of the square of all pixels in an image:

```
psImage *A, *B;
A = someCoolImage(); // Initialise A

B = (psImage*)psBinaryOp (NULL, A, "^", psScalarAlloc(2, PS_TYPE_S16));
// Note, have to cast the output to a psImage for proper compilation
psUnaryOp(B, B, "sin");
```

Note that the `psUnaryOp` is performed on `B` in-place.

The list of required operators for `psBinaryOp` are:

- Addition ($+$)
- Subtraction ($-$)
- Multiplication ($*$)
- Division ($/$)
- Power ($\hat{}$)

- Minimum (min)
- Maximum (max)
- Logical or (|; integer type only)
- Logical and (&; integer type only)

The list of required operators for `psUnaryOp` are:

- Absolute value (abs)
- Exponent (exp)
- Natural Log (ln)
- Power of 10 (ten)
- Log (log)
- Sine (sin and dsin)
- Cosine (cos and dcos)
- Tangent (tan and dtan)
- Arcsine (asin and dasin)
- Arccosine (acos and dacos)
- Arctan (atan and datan)

The trigonometric operators prefixed with "d" refer to the standard trigonometric operators acting on input that is in decimal degrees.

6.9 Matrix operations and linear algebra

In addition to the ability to perform image arithmetic (§6.8), we also require the ability to perform basic linear algebra on matrices, in order to solve equations. We use `psImage` as a matrix, since it has the correct form. We define the following basic matrix operations:

- LU Decompose a matrix, and solve for x in $Ax = b$;
- Invert a matrix;
- Calculate a matrix determinant;
- Perform matrix addition, subtraction and multiplication;
- Transpose a matrix;
- Get eigenvectors for a matrix; and

- Convert a matrix to a vector.

The corresponding APIs follow.

In the event of an error, the matrix functions shall return `NULL`.

```
psImage *psMatrixLUD(psImage *out, psVector **perm, const psImage *in);
psVector *psMatrixLUSolve(psVector *out, const psImage *LU, const psVector *RHS, const psVector *perm);
```

The above functions decompose a matrix, `in`, into its LU representation (`psMatrixLUD`, which returns the decomposed matrix), and uses the decomposed matrix to solve for x in the equation $Ax = b$. In this case, the LU decomposed matrix A is specified as `LU`, and b is `RHS`; the solution vector for x is returned. These functions are specified for data types `psF32, psF64`. The output and input vectors and images must all have the same data type.

The GSL routines require the use of a permutation vector, `perm`. This vector shall be created by `psMatrixLUD`, and the user need only pass this to `psMatrixLUSolve` before destroying it in the standard manner. In order to avoid memory leaks, `perm` must be `NULL` on calling `psMatrixLUD`.

```
psImage *psMatrixInvert(psImage *out, const psImage *in, float *determinant);
```

`psMatrixInvert` returns the inverse of the specified matrix (`in`), along with the `determinant`, if the `float` pointer is non-NULL. The input and output images must have the same type. This function is specified for data types `psF32, psF64`.

The matrix determinant may be calculated using `psMatrixDeterminant`, which simply returns the determinant for the specified matrix, `in`. This function is specified for data types `psF32, psF64`.

```
float psMatrixDeterminant(const psImage *in);
```

Matrix multiplication is supported through `psMatrixMultiply`. This function is specified for data types `psF32, psF64`.

```
psImage *psMatrixMultiply(psImage *out, const psImage *in1, const psImage *in2);
```

The transpose of an input matrix, `in`, is returned by `psMatrixTranspose`, optionally into the provided matrix `out`. This function is specified for data types `psF32,psF64`.

```
psImage *psMatrixTranspose(psImage *out, const psImage *in);
```

Eigenvectors of a matrix are calculated by `psMatrixEigenvectors`. This function is specified for data types `psF32, psF64`. Input and output data types should match.

```
psImage *psMatrixEigenvectors(psImage *out, const psImage *in);
```

Should this return an array of vectors? Specified here as currently implemented by MHPCC. (TBD)

Finally, we specify two functions to convert between matrices and vectors. This allows the use of vectors in matrix arithmetic, since a vector can be converted to a matrix, utilized, and the result can be converted back to a vector if required. This function is specified for data types `psF32, psF64`. Input and output data types should match.

```
psVector *psMatrixToVector(psVector *outVector, const psImage *inImage);
psImage *psVectorToMatrix(psImage *outImage, const psVector *inVector);
```

6.9.1 Sparse Matrices

Very large matrices (N elements > 1000) can be very time consuming to manipulate. A certain class of matrices, sparse matrices, are amenable to iterative solutions even when extremely large (100,000s of elements). The following functions define structures to efficiently represent sparse matricies, to add elements to those matrices, and to perform linear algebra with them.

```
typedef struct {
    psVector *Aij;
    psVector *Bfj;
    psVector *Qii;
    psVector *Si;
    psVector *Sj;
    int Nelem;
    int Nrows;
} psSparse;
```

The above structure contains the following elements, describing a sparse matrix equation of the form $A\bar{x} = \bar{B}f$:

- the vector `Aij` contains the populated elements of the matrix
- the vector `Bfj` contains the elements of the vector Bf
- the vector `Qii` contains the diagonal elements of Aij
- the vector `Si` contains the i-index values of Aij
- the vector `Sj` contains the j-index values of Aij
- the element `Nelem` defines the total number of elements in matrix `Aij`
- the element `Nrows` defines the size of the matrix `Aij`

The following structure defines constraints to limit the range of the value matrix equation solution:

```
typedef struct {
    double paramDelta;
    double paramMin;
    double paramMax;
} psSparseConstraint;
```

The following function allocates a sparse matrix structure:

```
psSparse *psSparseAlloc (int Nrows, int Nelem);
```

The following function adds a new matrix element. The user should only add elements above the diagonal.

```
bool psSparseMatrixElement(psSparse *sparse, int i, int j, float value);
```

The following function define a new sparse matrix equation vector element:

```
void psSparseVectorElement(psSparse *sparse, int i, float value);
```

The following function performs the operation "matrix times vector" on a sparse matrix and a vector:

```
psVector *psSparseMatrixTimesVector(psVector *output, const psSparse *matrix,
                                    const psVector *vector);
```

The following function re-sorts a sparse matrix to have all elements in index order rather than insertion order. The user must call this function before attempting to solve, but after populating, the matrix and vector:

```
bool psSparseResort(psSparse *sparse);
```

The following function iteratively solves the equation $A\bar{x} = \bar{B}f$. For the value of $\bar{x}$, a good starting guess is the vector $\bar{B}f$

```
psVector *psSparseSolve(psVector *output, psSparseConstraint constraint,
                        const psSparse *sparse, int Niter);
```

6.10 (Fast) Fourier Transforms

We require the ability to calculate the (fast) Fourier transforms of floating-point one-dimensional vectors and two-dimensional images. We expect that these will be implemented through wrapping an external library. We define the following APIs to support FFT operations on vectors:

```
psVector *psVectorFFT(psVector *out, const psVector *in, psFFTFlags direction);
psVector *psVectorReal(psVector *out, const psVector *in);
psVector *psVectorImaginary(psVector *out, const psVector *in);
psVector *psVectorComplex(psVector *out, const psVector *real, const psVector *imag);
psVector *psVectorConjugate(psVector *out, const psVector *in);
psVector *psVectorPowerSpectrum(psVector *out, const psVector *in);
```

The forward and reverse FFT is calculated using `psVectorFFT`, which takes as input the vector of interest (`in`) and the direction (`direction`), which is specified by an enumerated type defined below. The input vector may be of type `psF32` or `psC32`, the result is always `psC32`. If the input vector is `psF32`, the direction must be forward. Neither the forward or inverse transforms must multiply by $1/N$ (or $1/N^{1/2}$), and so it falls to the responsibility of the user to multiply a vector that has been forward- and reverse-transformed by $1/N$.

Conversions between complex and real vectors requires the functions `psVectorReal`, which returns the real part (`psF32`) of the complex vector `in`, `psVectorImaginary`, which returns the the magnitude of the imaginary part (`psF32`) of the complex vector `in`, and `psVectorComplex`, which constructs a complex vector (`psC32`) from the real and imaginary components `real` and `imag`, both of type `psF32`.

We also specify the additional utility functions `psVectorConjugate` and `psVectorPowerSpectrum` which construct the complex conjugate (`psC32`) and the magnitude (`psF32`) vectors of the input complex vector (`psC32`).

The direction of an FFT performed by `psVectorFFT` is specified by an enumerated type, `psFFTFlags`:

```
/** Specify direction of FFT, and if the result is real or not */
typedef enum {
```

```
    PS_FFT_FORWARD = 1,                 ///< psImageFFT/psVectorFFT should perform a forward FFT.
    PS_FFT_REVERSE = 2,                 ///< psImageFFT/psVectorFFT should perform a reverse FFT.
    PS_FFT_REAL_RESULT = 4              ///< psImageFFT/psVectorFFT should return a real image. This is val
                                        ///< for only reverse FFT, i.e., the psImageFFT/psVectorFFT flag
                                        ///< parameter should appear as PS_FFT_REVERSE+PS_FFT_REAL_RESULT.
} psFFTFlags;
```

The entry PS_FFT_REAL_RESULT means that the output of an inverse FFT is to be purely real. An example of its use is:

```
out = psImageFFT(in, PS_FFT_REVERSE | PS_FFT_REAL_RESULT);
```

The output from a FFT must be of the same size as the input, and the size of the power spectrum equal to $N/2 + 1$, where N is the number of elements in the input. In the future, if this adversely affects performance, this will be revised so that the redundant information will be neglected.

In analogy with the vector FFT operations, we also define matching image operations as follows:

```
psImage *psImageFFT(psImage *out, const psImage *image, psFFTFlags direction);
psImage *psImageReal(psImage *out, const psImage *in);
psImage *psImageImaginary(psImage *out, const psImage *in);
psImage *psImageComplex(psImage *out, const psImage *real, const psImage *imag);
psImage *psImageConjugate(psImage *out, const psImage *in);
psImage *psImagePowerSpectrum(psImage *out, const psImage *in);
```

6.11 Convolution

Convolution will be an essential operation for the IPP. For example, if images on the sky are obtained in Orthogonal Transfer (OT) mode, then the calibration frames used to correct them must be convolved by a kernel derived from the list of OT shifts made during the exposure. Also, convolution is also required for object detection and PSF-matching.

6.11.1 Basic Image Smoothing

```
bool psImageSmooth(psImage *image, double sigma, double Nsigma);
```

The quickest way to smooth an image is to smooth by parts, using 1D gaussian smoothing in X and Y independently. `psImageSmooth` applies a single parameter Gaussian (circularly symmetric) by smoothing first in the X and then in the Y directions with just a vector. In general, for a Gaussian of dimension N, this process is 2N faster than for the 2D convolution defined below. The arguments to this function include the image to be smoothed, the width of the smoothing kernel in pixels (`sigma`), and the size of the smoothing box in sigmas.

6.11.2 Kernel definition

In order to perform a convolution, we need to define the convolution kernel. We need a more general object than a `psImage` so that we can incorporate the offset from the $(0, 0)$ pixel to the $(0, 0)$ value of the kernel. It might be convenient to allow both positive and negative indices to convey the positive and negative shifts. One might consider setting the `x0` and `y0` members of a `psImage` to the appropriate offsets, but this is not the purpose of these members, and doing so may affect the behavior of other `psImage` operations. We define a `psKernel`:

```
/** A convolution kernel */
typedef struct {
    psImage *image;                     ///< Kernel data, in the form of an image
    int xMin;                           ///< Most negative indices
    int yMin;                           ///< Most negative indices
    int xMax;                           ///< Most positive indices
    int yMax;                           ///< Most positive indices
    float **kernel;                     ///< Pointer to the kernel data
    float **p_kernelRows;               ///< Pointer to the rows of the kernel data
} psKernel;
```

The kernel data is carried primarily by the `data` member which is a normal `psImage`. In order to allow negative indices, we add two additional members. `kernelRows` is an array of pointers to `float`; these pointers point into the `psImage` data, offset by the desired column offset. `kernel` point into the `kernelRows`, offset by the desired row offset. This construction allows the `kernel` member to use negative indices, while preserving the location of `psMemBlock`s relative to allocated memory.

The maximum extent of the kernel shifts shall be defined by the `xMin`, `xMax`, `yMin` and `yMax` members. Note that `xMin` and `yMin`, under normal circumstances, should be negative numbers. That is, `myKernel->kernel[-3][-2]` may be defined if `yMin` and `xMin` are equal to or more negative than -3 and -2, respectively.

Of course, we require the appropriate constructor:

```
psKernel *psKernelAlloc(int xMin, int xMax, int yMin, int yMax);
```

`psKernelAlloc` shall allocate a kernel. In the event that one of the minimum values is greater than the corresponding maximum value, the function shall generate a warning, and the offending values shall be exchanged.

6.11.3 Generation of a convolution kernel

Given a list of values (e.g., shifts made in the course of OT guiding), `psKernelGenerate` shall return the appropriate kernel. The API shall be the following:

```
psKernel *psKernelGenerate(const psVector *tShifts, const psVector *xShifts,
                           const psVector *yShifts, bool relative);
```

The vectors `xShifts` and `yShifts`, which are a list of shifts made at the times `tShifts`, are used to construct the appropriate kernel. If `relative` is `true`, then each shift is to be interpreted relative the shift made before; if `relative` is `false`, then the shifts are to be interpreted relative to some starting point. The elements of the vectors should be of an integer type; otherwise the values shall be truncated to integers. The output kernel shall be normalized such that the sum over the kernel is unity.

If the vectors are not all of the same number of elements, then the function shall generate an error and return `NULL`.

6.11.4 Convolve an image with the kernel

Given an input image and the convolution kernel, `psImageConvolve` shall convolve the input image, `in`, with the kernel, `kernel` and return the convolved image, `out`. The API shall be the following:

```
psImage *psImageConvolve(psImage *out, const psImage *in, const psKernel *kernel, bool direct);
```

Two methods shall be available for the convolution: if `direct` is `true`, then the convolution shall be performed in real space (appropriate for small kernels); otherwise, the convolution shall be performed using Fast Fourier Transforms (FFTs; appropriate for larger kernels). The latter option involves padding the input image, copying the kernel into an image of the same size as the padded input image, performing an FFT on each, multiplying the FFTs, and performing an inverse FFT before trimming the image back to the original size.

In the event that `out` is `NULL`, a new `psImage` shall be allocated and returned.

6.12 Random Numbers

Many applications involve random numbers, with the particular distribution depending upon the application. We will wrap the GSL implementation.

We define a `psRandom` type, which will allow us to carry around pre-computed random numbers, if required. For the time being, we only specify a single random number generator (RNG) in `psRandomType` (that provided by `gsl_rng_taus`), but we leave this open to expansion in the future, depending upon user requirements.

```
typedef enum {
    PS_RANDOM_TAUS                      ///< A maximally equidistributed combined Tausworthe generator
} psRandomType;

typedef struct {
    psRandomType type;                  ///< The type of RNG
    gsl_rng *gsl;                       ///< The RNG itself
} psRandom;
```

We require the ability to seed the random number generator (RNG) with a known value, so that bugs in modules which rely upon random numbers may be reproduced.

```
psRandom *psRandomAlloc(psRandomType type, unsigned long seed);
void psRandomReset(psRandom *rand, unsigned long seed);
```

`psRandomAlloc` shall construct a new instance of `psRandom` of the given `type`, and seed it with the given `seed`. The `seed` is specified as an `unsigned long` so that the system clock may be used to set it. If the `seed` is zero, then the RNG shall be seeded from `/dev/urandom` if it exists, or otherwise from the system clock, and the particular seed shall be printed using `psLogMsg` with a level of `PS_LOG_INFO`.

```
double psRandomUniform(const psRandom *r);
double psRandomGaussian(const psRandom *r);
double psRandomPoisson(const psRandom *r, double mean);
```

`psRandomUniform` shall return random numbers uniformly distributed on $[0, 1)$, using `gsl_rng_uniform`. `psRandomGaussian` shall return random numbers distributed on a Gaussian deviate, $N(0, 1)$, using `gsl_ran_gaussian`. `psRandomPoisson` shall return random numbers distributed on a Poisson distribution with the given `mean` using `gsl_ran_poisson`.

6.13 Ellipse Shape Functions

Astronomical objects are frequently decomposed into components represented by a radial function modified by an elliptical contour. There are a few ways in which the elliptical shape information can be represented depending on the circumstance in which it is used. The structures and functions in the section provide tools for converting between the elliptical representations. We provide three structures representing three ways in which the elliptical shape is represented. Like the `psRegion`, these datatypes and their supporting functions do not use allocators.

This structure represents an ellipse by its major and minor axis lengths and the orientation angle.

```
typedef struct {
    double major;
    double minor;
    double theta;
} psEllipseAxes;
```

This structure represents an elliptical Gaussian by the second moments measured for that Gaussian.

```
typedef struct {
    double x2;
    double y2;
    double xy;
} psEllipseMoments;
```

This structure represents an ellipse by the components of the cartesian coordiante equation: $(s_x x)^2 + (s_y y)^2 + s_{x,y} xy = R$

```
typedef struct {
    double sx;
    double sy;
    double sxy;
} psEllipseShape;
```

The following functions provide conversions between the elliptical shape representations:

```
psEllipseAxes psEllipseMomentsToAxes(psEllipseMoments moments);
psEllipseShape psEllipseAxesToShape(psEllipseAxes axes);
psEllipseAxes psEllipseShapeToAxes(psEllipseShape shape);
```

7 Astronomy-Related Functions

The previous sections of this document defined basic functionality needed by a wide range of scientific programming. The rest of this document concentrates on aspects of the library which are specific to astronomy. Some basic, relatively simple astronomy-specific functions are required which will serve as the foundation for building the higher level modules. These functions are not expected to cover every foreseeable function, but will serve as the building blocks of more complicated processing functions.

We require functions covering each of the following areas:

- Dates and times
- Detector and sky positions
- Astrometry
- Photometry
- Astronomical objects

7.1 Dates and times

7.1.1 Overview

We require a collection of functions to manipulate time data. These operations primarily consist of conversions between specific time formats. Internally, PSLib handles times as a structure similar to the POSIX `struct timeval` struct which has been extended to track the time system being represented.

7.1.2 Initialization and Finalization

The conversions between various time standards, and the polar motion requires importing external data ("time tables"), the locations of which are specified in a configuration file.

```
void psTimeInitialize(const char *timeConfig);
```

`psTimeInitialize` shall read the `psTime` configuration file (`timeConfig`) and set up the appropriate `psTimeTable`s and predictions.

Calls to `psTime` functions that require the time tables before calling `psTimeInitialize` shall result in an error.

```
void psTimeFinalize(void);
```

`psTimeFinalize` shall free memory that was allocated by `psTime` functions, allowing a subsequent search for leaked memory (even memory marked as `persistent`).

7.1.3 Data Types

```
typedef enum {
    PS_TIME_TAI,      ///< seconds since 1970-01-01T00:00:00Z (Gregorian), SI seconds
    PS_TIME_UTC,      ///< seconds since 1970-01-01T00:00:00Z (Gregorian), SI seconds + leapseconds
    PS_TIME_UT1,      ///< seconds since 1970-01-01T00:00:00Z (Gregorian), variable length seconds
    PS_TIME_TT,       ///< seconds since 1970-01-01T00:00:00Z (Gregorian), SI seconds
} psTimeType;

typedef enum {
    PS_IERS_A,                    ///< IERS Bulletin A
    PS_IERS_B,                    ///< IERS Bulletin B
} psTimeBulletin;

typedef struct {
    psS64            sec;         ///< seconds, negative values represent dates before 1970
    psU32            nsec;        ///< nanoseconds
    bool             leapsecond;  ///< if time falls on a UTC leapsecond
    psTimeType       type;        ///< type of time
} psTime;
```

7.1.4 Constructors

To allocate a new, initialized to zero, `psTime`:

```
psTime *psTimeAlloc(psTimeType type);
```

To allocate a new `psTime` set to the current time (in the given system):

```
psTime *psTimeGetNow(psTimeType type);
```

7.1.5 Time Conversion

Converting between the `psTime` time systems is done with:

```
psTime *psTimeConvert(psTime *time, psTimeType type);
```

This function may be used to convert between the various `psTimeType` time representations. The `time` is modified and returned. Conversion between all of the *SI* length second systems should be implemented by first converting to TAI and then to the destination system. UT1 is a special case for conversion as it uses variable length seconds. Conversation to UT1, via TAI, is allowed but conversion *from* UT1 is currently forbidden.

The following function converts to Local Mean Sidereal Time. It is necessary to provide the local longitude (specified in radians, positive East of Greenwich) as well:

```
double psTimeToLMST(psTime *time, double longitude);
```

The function may accept any of the `psTimeType` types with *SI* length seconds. The `time` is modified (rationlized to TAI units) and returned. Note that this function requires the value $UT1 - UTC$ (see `psTimeGetUT1Delta()`) and should use the UT1 values interpolated from IERS bulletin B.

The following utility function encapsulates the PSLib mechanism to extract the value of $UT1 - UTC$ from the IERS Time Tables:

```
double psTimeGetUT1Delta(const psTime *time, psTimeBulletin bulletin);
```

The following function provides tidal corrections to UT1-UTC, using the Ray model of Simon et al (REF).

```
psTime *psTime_TideUT1Corr(const psTime *time);
```

Leap seconds are added to UTC in order to keep it within $0.9s$ of UT1 (which is defined relative to the Earth's rotation, and hence is useful for astronomical purposes). The following function calculates the absolute number of leap seconds different between two times.

```
long psTimeLeapSecondDelta(const psTime *time1, const psTime *time2);
```

The following function accepts `PS_TIME_UTC` objects and determines if the time is potentially a leapsecond, returning `TRUE` if so.

```
bool psTimeIsLeapSecond(const psTime *utc);
```

7.1.6 External Date and Time Formats

A collection of functions convert from the `psTime` types to various external formats. The ISO8601 format[2] we will be using is "YYYY-MM-DDThh:mm:ss.sZ". *Note that the ISO8601 format is only valid for the years* $0000..9999$. *Values out side that range should cause a* `NULL` *pointer to be returned.*

```
double psTimeToJD(const psTime *time);
double psTimeToMJD(const psTime *time);
psString psTimeToISO(const psTime *time);
struct timeval *psTimeToTimeval(const psTime *time);
struct tm *psTimeToTM(const psTime *time);
psString *psTimeStrftime(const psTime *time, const char *format);
```

The `psTimeToISO()` function converts `PS_TIME_UTC` objects with the `leapsecond` flag set to represent the number of seconds as ":60".

The `psTimeStrftime()` function converts psTime objects into a string who's formatted is defined by `format`. Where `format` is a `strftime(3)` compatable formating string.

[2] ISO8601:2000(E) - http://www.pvv.ntnu.no/ nsaa/8601v2000.pdf

7.1.7 Date and Time Parsing

A related collection of functions convert from external formats to a `psTime` type.

```
psTime *psTimeFromJD(double jd);
psTime *psTimeFromMJD(double mjd);
psTime *psTimeFromISO(const char *input, psTimeType type);
psTime *psTimeFromTimeval(const struct timeval *input);
psTime *psTimeFromTM(const struct tm* time);
psTime *psTimeFromUTC(psS64 sec, psU32 nsec, bool leapsecond);
psTime *psTimeFromTT(psS64 sec, psU32 nsec);
psTime *psTimeStrptime(const char *s, const char *format);
```

Where `psTimeFromUTC()` will validate that it is possible for the input time to land on a UTC leapsecond (using `psTimeIsLeapSecond()`) if `leapsecond` is set to `true`.

The `psTimeStrptime()` function parses the string `s` into a `psTime` using a `strptime(3)` compatible format specified as `format`.

7.1.8 Date and Time Math

```
psTime *psTimeMath(const psTime *time, double delta);
double psTimeDelta(const psTime *time1, const psTime *time2);
```

`psTimeMath` adds the `delta` (in seconds; may be negative) to a `psTime`. `PS_TIME_UTC` objects will be converted to `PS_TIME_TAI`, before applying the delta, and then back to `PS_TIME_UTC`. `psTimeDelta` gives the difference between times `time1` and `time2` (which may be negative).

The API documentation should note that when handling UT1, date math is allowed on the UT1 system but that the seconds are not of *SI* length (this is why conversion from UT1 is correctly forbidden). It should also be noted that with `psTimeDelta()` it is possible to overflow the dynamic range of a `psS64` (the type of `psTime.sec`).

Note that in both these functions the difference between two times is the elapsed number of seconds, including leap seconds. For example, if we add 30 seconds to 1998-12-31T23:59:45Z, the result is 1999-01-01T00:00:14Z, since a leap second was introduced at 1999-01-01T00:00:00Z.

Time math may only be done on `psTime` objects of the same type, and except in the case of UT1, the functions shall internally convert the `psTime` inputs to TAI before performing the math; this ensures that leap seconds are correctly counted.

The type of the time returned by `psTimeMath` shall be the same as that of the input `time`.

7.1.9 Time Tables

The offset of UTC from UT1, $\Delta UT = UT1 - UTC$, as well as the polar motion, x_p and y_p, may be determined from table lookups. Tables are available covering different time periods and with different time resolution, and so it is important to be able to utilize multiple tables. Some tables may be found at:

- IERS Bulletin A & B (1 year ago $\rightarrow$ now + $\sim$3 months)

  - `ftp://maia.usno.navy.mil/ser7/finals.daily`
- IERS Bulletin A & B (1973-1-2 → now + ∼1 year)
  - `ftp://maia.usno.navy.mil/ser7/finals.all`
- $TAI - UTC$
  - `ftp://maia.usno.navy.mil/ser7/tai-utc.dat`
- $TAI - UTC$ (contains estimates prior to 1972)
  - `http://hpiers.obspm.fr/eoppc/eop/eopc01/eopc01.1900-2004`

The format of `finals.daily` and `finals.all` is documented at `ftp://maia.usno.navy.mil/ser7/readme.final`

The tables shall reside on local disk in known locations (i.e., there is no need that they are downloaded from the internet and parsed by PSLib). The format of these files shall be simple, for speed in reading. Each line shall contain the date in MJD and the following values from both the A & B IERS bulletins: x_p (in arcseconds), y_p (in arcseconds) and ΔUT (in seconds). This format must be readable by `psLookupTableRead`. For example:

```
# finals.daily, 2005-03-17
# ftp://maia.usno.navy.mil/ser7/finals.daily
# MJD    PM-IP  PM-X"   PM-Y"   UT1-YP  UT1-UTCs    PM-X"   PM-Y"   UT1-UTCs
#        A      A       A       A       A           B       B       B
53403.00 I      .089198 .207785 I       -.5218847   .089220 .207360 -.5219040
53404.00 I      .086549 .206873 I       -.5224164   .086670 .206850 -.5224260
53405.00 I      .083589 .206143 I       -.5227681
53406.00 I      .081541 .205865 I       -.5229582
```

The location of these files, their priority order, and the "from" and "to" dates of applicability will be specified through metadata (§7.1.9.1).

Calls to `psTime` functions that require the time tables before calling `psTimeInitialize` shall result in an error.

When a value is required, the tables shall shall be checked in priority order to see if the date is within the range of applicability for the table. If a table is found that is applicable, then the appropriate value shall be derived from linear interpolation between the nearest entries in the table. If no table is found that is applicable, and the required date is later than those covered in the tables, a warning shall be generated, and a pre-determined formula shall be applied (§7.1.9.1). For dates prior to those covered in the tables, the function shall generate a warning and use pre-determined values (§7.1.9.1).

7.1.9.1 Time Metadata

The following metadata keys will be used in time calculations:

Metadata key	Purpose
psLib.time.tables.dir	Time table directory
psLib.time.tables.files	Time table file names (space-delimited)
psLib.time.tables.from	Time tables are valid from these MJDs (vector)
psLib.time.tables.to	Time tables are valid to these MJDs (vector)
psLib.time.before.xp	Value of XP for before the earliest MJD
psLib.time.before.yp	Value of YP for before the earliest MJD
psLib.time.before.dut	Value of UT1-UTC for before the earliest MJD
pslib.time.predict.xp	A vector containing the x_p prediction formula coefficients
pslib.time.predict.yp	A vector containing the y_p prediction formula coefficients
pslib.time.predict.mjd	A value containing the MJD offset for temporary variables
pslib.time.predict.dut	A vector containing the UT1-UTC prediction formula coefficients

These metadata keys shall reside in a configuration file (§3.6.4), `psTime.config`, which shall be loaded into a `psMetadata` structure private to `psTime`, as part of `psLibInit`. An example of `psTime.config` follows:

```
# This configuration file specifies values required for time calculations by psLib.
psLib.time.tables.dir   STR     /home/panstarrs/psLib/config/                   # Directory for time tables
psLib.time.tables.n     U8      2                                               # Number of time tables
psLib.time.tables.files STR     bulletinA_09Sep2004.dat eopc01_1900_2004.40.dat # These are the file names
@psLib.time.tables.from F64     53258.0, 15020.0                                # Valid from these MJDs
@psLib.time.tables.to   F64     53622.0, 53258.0                                # Valid to these MJDs
psLib.time.before.xp    F64     0.0                     # Value of XP for before the earliest MJD
psLib.time.before.yp    F64     0.0                     # Value of YP for before the earliest MJD
psLib.time.before.dut   F64     0.0                     # Value of UT1-UTC for before the earliest MJD

# Now follows formulae for predicting ahead of the most recent available table entry.
# xp = [0] + [1]*cos A + [2]*sin A + [3]*cos C + [4]*sin C
# yp = [0] + [1]*cos A + [2]*sin A + [3]*cos C + [4]*sin C
# A = 2*pi*(MJD - pslib.time.predict.mjd)/365.25
# C = 2*pi*(MJD - pslib.time.predict.mjd)/435.0
# dut = ut1-utc = [0] + [1]*(MJD - [2]) - (UT2-UT1)
# ut2-ut1 = 0.022 sin(2*pi*T) - 0.012 cos(2*pi*T) - 0.006 sin(4*pi*T) + 0.007 cos(4*pi*T)
# T = 2000.0 + (MJD - 51544.03)/365.2422
@psLib.time.predict.xp  F64     0.0569, 0.0555, -0.0200, 0.0513, 0.1483
@psLib.time.predict.yp  F64     0.3498, -0.0176, -0.0498, 0.1483, -0.0513
psLib.time.predict.mjd  F64     53257.0
@psLib.time.predict.dut F64     -0.4944, -0.00023, 53262.0
```

A series of test inputs is contained in §B.

7.2 Timers

It is useful to be able to time an operation. For this purpose, we specify the following:

```
bool psTimerStart(char *name);
psF64 psTimerMark(char *name);
psF64 psTimerClear(char *name);
psF64 psTimerStop(void);
```

`psTimerStart` shall store the current time in a `psHash` of timers, under the supplied `name`. `psTimerMark` shall return the elapsed time, in seconds, for the timer specified by `name`. `psTimerClear` resets the named timer. `psTimerStop` shall free all memory associated with all timers, so that no memory is leaked, and return the maximum time expended.

7.3 Linear and Spherical Coordinates

Both detector and sky positions will be used extensively in the IPP. The first are linear coordinates which conform to Euclidean geometry while the second are angular coordinates which define a position on the sphere of the sky. We put these into two structures, `psPlane` and `psSphere`, respectively. Partitioning these two will enable error-checking. An alternative representation for angular positions is the 3-D unit vector. These are used in particular as part of spherical rotation calculations. We define `psCube` to represent such an element.

```
typedef struct {
    double x;                          ///< x position
    double y;                          ///< y position
    double xErr;                       ///< Error in x position
    double yErr;                       ///< Error in y position
} psPlane;

typedef struct {
    double r;                          ///< RA
    double d;                          ///< Dec
    double rErr;                       ///< Error in RA
    double dErr;                       ///< Error in Dec
} psSphere;

typedef struct {
    double x;                          ///< cos (DEC) cos (RA)
    double y;                          ///< cos (DEC) sin (RA)
    double z;                          ///< sin (DEC)
    double xErr;                       ///< Error in x
    double yErr;                       ///< Error in y
    double zErr;                       ///< Error in z
} psCube;
```

Three major classes of coordinate transformations are necessary. First, linear coordinates from one frame must be converted to linear coordinates in a different frame of references. Simple transformations of this type are independent of other quantities of the positions – they are simply mapping between two linear spaces. In practice, these transformations may often be a function of the color or even magnitude of the imaged object. The second type of conversion is the transformation of linear coordinates to angular coordinates and vice-versa. This conversion depends on the desired projection, and may represent the real mapping performed by the telescope or may simply represent a convenient mechanism to display 3D coordinates in useful forms. The third conversion of interest is the transformation of one set of spherical coordinates to another set. Frequently in astronomy, these conversions consist only of rotations between the two spherical coordinates systems, where the coordinates of the pole and equatorial rotation between the two systems define the transformation. Conversions between standard coordinate systems such as Galactic, Ecliptic, and various epochs of the Celestial coordinates are represented by these spherical transformations.

Constructors for these are straight-forward:

```
psPlane  *psPlaneAlloc(void);
psSphere *psSphereAlloc(void);
psCube   *psCubeAlloc(void);
```

Initialization of the structures is not necessary.

7.3.1 Linear Coordinate Transformations

We specify two types of transforms between coordinate systems. The first consists simply of two 2D polynomials to transform both components – the output coordinates depend only on the input coordinates and no other quantities of objects at those coordinates. The second consists of two 4D polynomials in which the output coordinates are also specified to be a function of the color and magnitude of the object with the given coordinates. This type of coordinate transformation is necessary to represent the (color-dependent) optical distortions caused by the atmosphere and camera optics, and the possibly effects of charge transfer inefficiency. We specify two structures to represent the coefficients of these transformations:

```
typedef struct {
    psPolynomial2D *x;
    psPolynomial2D *y;
} psPlaneTransform;
```

The `psPolynomial2D` structures represent polynomials of arbitrary order as a function of two dimensions. There is one of these structures for each of the two output dimensions. As an example, consider the simple transformation from one linear coordinate frame (x, y), e.g., on a CCD, to a second frame (p, q), e.g., on a chip. If we have only first order terms in the transformation `psPlaneTransform T`, the new coordinates would be represented by the terms:

```
p = T.x->coeff[0][0] + x*T.x->coeff[1][0] + y*T.x->coeff[0][1];
q = T.y->coeff[0][0] + x*T.y->coeff[1][0] + y*T.y->coeff[0][1];
```

where we have excluded the basic cross-term ($x \times y$) by using the mask: `T.x->mask[1][1] = 1; T.y->mask[1][1] = 1;`

The `psPlaneDistort` represents an optical distortion. The lowest two terms are the x and y axis of the target system. The higher two terms may represent color and magnitude terms.

```
typedef struct {
    psPolynomial4D *x;
    psPolynomial4D *y;
} psPlaneDistort;
```

Like `psPlaneTransform`, `psPlaneDistort` contains two `psPolynomial4D` structures representing polynomials of arbitrary order as a function of four, rather than two dimensions. There is one of these structures for each of the two output dimensions. In this structure, the highest two dimensions could represent a color and magnitude. As an example, consider the simple transformation from one linear coordinate frame (x, y), e.g., on a CCD, of an object with color and magnitude (c, m) to a second frame (p, q), e.g., the focal plane. If we have only first order terms in the transformation `psPlaneDistort T`, the new coordinates would be represented by the terms:

```
p = T.x->coeff[0][0][0][0] + x*T.x->coeff[1][0][0][0] + y*T.x->coeff[0][1][0][0]
  + c*T.x->coeff[0][0][1][0] + m*T.x->coeff[0][0][0][1]
q = T.y->coeff[0][0][0][0] + x*T.y->coeff[1][0][0][0] + y*T.y->coeff[0][1][0][0]
  + c*T.y->coeff[0][0][1][0] + m*T.y->coeff[0][0][0][1]
```

where we have again excluded the cross-term ($x \times y$) by using the mask.

Each of `psPlaneTransform` and `psPlaneDistort` has an appropriate allocator that takes the polynomial order in each dimension. Both the `x` and `y` polynomials shall be have the same dimensions.

```
psPlaneTransform *psPlaneTransformAlloc(int order1, int order2);
psPlaneDistort *psPlaneDistortAlloc(int order1, int order2, int order3, int order4);
```

We require corresponding functions to apply the transformations to a specified coordinate `coords`:

```
psPlane *psPlaneTransformApply(psPlane *out,
                               const psPlaneTransform *transform,
                               const psPlane *coords);
psPlane *psPlaneDistortApply(psPlane *out,
                             const psPlaneDistort *distort,
                             const psPlane *coords,
                             float mag, float color);
```

The following functions perform operations on transformations:

```
psPlaneTransform *psPlaneTransformInvert(psPlaneTransform *out, const psPlaneTransform *in,
                                         psRegion region, int nSamples);
psPlaneTransform *psPlaneTransformCombine(psPlaneTransform *out, const psPlaneTransform *trans1,
                                          const psPlaneTransform *trans2, psRegion region,
                                          int nSamples);
bool psPlaneTransformFit(psPlaneTransform *trans, const psArray *source, const psArray *dest,
                         int nRejIter, float sigmaClip);
```

`psPlaneTransformInvert` shall return the transformation that inverts the given transformation. It may assume that the input transformation is one-to-one, and that the inverse transformation may be specified through using polynomials of the same type and order as the forward transformation. In the event that the input transformation is linear, an exact solution may be calculated; otherwise `nSamples` samples in each axis, covering the region specified by `region` shall be used as a grid to fit the best inverse transformation. The function shall return `NULL` if it was unable to generate the inverse transformation; otherwise it shall return the inverse transformation. In the event that `out` is `NULL`, a new `psPlaneTransform` shall be allocated and returned. **is this subimage-safe? (TBD)**

`psPlaneTransformCombine` takes two transformations (`trans1` and `trans2`) and returns a single transformation that has the effect of performing `trans1` followed by `trans2`. In the event that the input transformation is linear, an exact solution may be calculated; otherwise `nSamples` samples in each axis, covering the region specified by `region` shall be used as a grid to fit the best inverse transformation. The function shall return `NULL` if it was unable to generate the transformation; otherwise it shall return the transformation.

`psPlaneTransformFit` takes two arrays containing matched coordinates (i.e., coordinates in the `source` array correspond to the coordinates in the `dest` array) and returns the best-fitting transformation. The `source` and `dest` will contain `psCoord`s. In the event that the number of coordinates in each is not identical, the function shall generate a warning, and extra coordinates in the longer of the two shall be ignored. The `trans` transform may not be `NULL`, since it specifies the desired order, polynomial type and any polynomial terms to mask. `nRejIter` rejection iterations shall be performed, wherein coordinates lying more than `sigmaClip` standard deviations from the fit shall be rejected.

7.3.1.1 Derivatives

In order to simply calculate which pixels in one coordinate frame overlap those in another coordinate frame tied by a `psPlaneTransform`, we need to calculate the derivative of a transformation with respect to each coordinate.

```
psPlane *psPlaneTransformDeriv(psPlane *out, const psPlaneTransform *transformation, const psPlane *coord);
```

`psPlaneTransformDeriv` shall return the derivatives of the `transformation` at the specified coordinates, `coord`. Both the derivatives with respect to x and y shall be returned. If `out` is non-`NULL`, it shall be modified and returned; otherwise a new `psPlane` shall be allocated and returned. In the event that either `transformation` or `coord` are `NULL`, the function shall generate an error and return `NULL`.

```
psPixels *psPixelsTransform(psPixels *out, const psPixels *input, const psPlaneTransform *inToOut);
```

`psPixelsTransform` shall generate a list of pixels in the output coordinate frame that overlap the `input` pixels in the input coordinate frame through the specified transformation, `inToOut`. Note that this is more complicated than simply transforming the `input` pixels, but requires the evaluation of the derivatives of the transformation in order to obtain the list of all pixels in the output coordinate frame that possibly overlap the pixel in the input coordinate frame (assuming that $x' + \Delta x' = f(x) + \Delta x \times \partial f(x)/\partial x$, where x' is the coordinate in the output coordinate frame, $\Delta x'$ is the size of the region in the output coordinate frame in the positive direction that overlaps the input pixel, x is the coordinate in the input coordinate frame, $f(x)$ is the transformation from the input to the output coordinate frames, and Δx is the size of a pixel; care should be taken with half-pixel problems). In the event that `input` or `inToOut` are `NULL`, the function shall generate an error and return `NULL`.

7.3.2 Spherical Rotations

Spherical rotations represent coordinate transformation in 3-D, as well as the effects of precession and nutation. We need spherical rotations to convert between ICRS, Galactic and Ecliptic coordinates, and to determine Alt-Az coordinates for sources. All of these basic spherical transformations represent rotations of the spherical coordinate reference. We specify a general transformation function which takes a structure, `psSphereRot`, defining the transformation between two spherical coordinate systems. The structure contains the elements of a quaternion to represent the spherical rotational. We define two allocators for `psSphereRot`, one which defines the rotation in terms of the coordinate of the pole and the rotation about that pole. The other defines the rotation from the elements of the quaternion. We also specify functions to manipulate `psSphereRot` in several useful way.

```
typedef struct {
    double q0;
    double q1;
    double q2;
    double q3;
} psSphereRot;
```

The constructor is defined as follows:

```
psSphereRot *psSphereRotAlloc(double alphaP, double deltaP, double phiP);
```

where `alphaP` and `deltaP` define the coordinates in the input system of the axis of rotation (the north pole of the output system), while `phiP` defines the rotation about that pole. This last angle is also equal to 270°- ϕ_a, where ϕ_a is the longitude in the output system of the ascending node (equatorial intersection between the two systems, e.g, the first point of Ares).

The `psSphereRot` may also be constructed by supplying the elements of the quaternion to the following function:

```
psSphereRot *psSphereRotQuat(double q0, double q1, double q2, double q3);
```

This function normalizes the quaternion, so the input elements need not be normalized.

Spherical coordinates may be transformed by providing the transformation and the coordinate in the input system to psSphereRot. The output pointer may be optionally supplied, or if NULL, is allocated by the function.

```
psSphere *psSphereRotApply(psSphere *out, const psSphereRot *transform, const psSphere *coord);
```

The following function combines two rotations, to produce a single rotation which is the equivalent of applying the first rotation and then the second. The output rotation may be supplied, or will be allocated if NULL.

```
psSphereRot *psSphereRotCombine(psSphereRot *out, const psSphereRot *rot1, const psSphereRot *rot2)
```

The following function changes the given rotation to its inverse:

```
psSphereRot *psSphereRotInvert(psSphereRot *rot)
```

The 3-vector representation of the angles (psCube) is needed to implement these functions, and is useful in other circumstances as well. Two utility functions are provided to convert between the angular and 3-vector representations:

```
psSphere *psCubeToSphere(const psCube *cube);
psCube *psSphereToCube(const psSphere *sphere);
```

7.3.3 Offsets

We require a function to calculate the offset between two positions on the sky, as well as a function to apply an offset to a position. The first determines the offset (RA,Dec) on the sky between two positions. The second applies the given offset to the coordinate. Both an offset mode and an offset unit may be defined. The mode may be either PS_SPHERICAL, in which case the specified offset corresponds to an offset in angles, or it may be PS_LINEAR, in which case the offset corresponds to a linear offset in a local projection. The offset unit may be in one of PS_ARCSEC, PS_ARCMIN, PS_DEGREE, and PS_RADIAN, which specifies the units of the offset only.

```
psSphere *psSphereGetOffset(const psSphere *position1,
                            const psSphere *position2,
                            psSphereOffsetMode mode,
                            psSphereOffsetUnit unit);

psSphere *psSphereSetOffset(const psSphere *position,
                            const psSphere *offset,
                            psSphereOffsetMode mode,
                            psSphereOffsetUnit unit);

typedef enum {
    PS_SPHERICAL,                       ///< Offset on a sphere
    PS_LINEAR                           ///< Linear offset
} psSphereOffsetMode;

typedef enum {
    PS_ARCSEC,                          ///< Arcseconds
    PS_ARCMIN,                          ///< Arcminutes
    PS_DEGREE,                          ///< Degrees
    PS_RADIAN                           ///< Radians
} psSphereOffsetUnit;
```

Note that these should propagate the errors appropriately.

7.4 Celestial Coordinate Systems

The following functions simply return the appropriate `psSphereRot` to convert between predefined spherical coordinate systems (i.e., ICRS, Ecliptic and Galactic). These are constructors as well as the above `psSphereRotAlloc`.

```
psSphereRot *psSphereRotICRSToEcliptic(const psTime *time);
psSphereRot *psSphereRotEclipticToICRS(const psTime *time);
psSphereRot *psSphereRotICRSToGalactic(void);
psSphereRot *psSphereRotGalacticToICRS(void);
```

7.5 Earth Orientation Calculations

One of the critical sets of calculations in astronomy is the sequence of steps needed to convert between the celestial coordinates of an object and the observed coordinates of the object. This problem is best divided into two major components: transformation between the celestial sphere and coordinates relative to the surface of the solid earth, excluding the effects of the atmosphere, and compensation for the effects of the atmosphere. In this section, we address the first of these two transformations: the Earth Orientation Calculations.

The Earth Orientation Calculations are further subdivided into several steps, illustrated in Figure 2 . Celestial coordinates are defined in the International Celestial Reference System (ICRS), which has the solar barycenter as its reference position and velocity. The next coordinate system is the Geocentric Celestial Reference System (GCRS), which uses the earth barycenter as a reference. The transformation between these two includes the aberration due to the Earth's velocity, the parallax of the object, which depends on both the Earth's position and the distance to the object of interest, and the general relativistic correction for the bending of light as it approaches the Earth.

The next set of transformations compensate for the 3-D rotation of the Earth on various timescales, including the effects of precession, nutation, and simple solid-body rotation. These calculations can be performed using different amounts of information for higher levels of precision. Since the Earth's rotation is constantly affected by stochastic processes (weather, earthquakes, etc), these conversions are constantly modified by observations reported by authoritative sources such as the US Naval Observatory. The target of this transformation is the International Terrestrial Reference System (ITRS), which is fixed with respect to the Earth's crust. This transformation is subdivided into slow precession and nutation (yielding the coordinate system CIP/CEO), followed by the Earth's rotation (yielding the coordinate system CIP/TEO), and finally corrections for the short-period motion of the Earth's pole.

7.5.1 Transformation from ICRS to GCRS

we need a function to construct the direction and speed elements given the time (TBD) .

supply the velocity as an un-normalized 3 vector? (TBD)

MHPCC: please code this section as currently specified. We will define a function, and algorithm, to return the current velocity vector given a time and position, which can be fed to this function (TBD) .

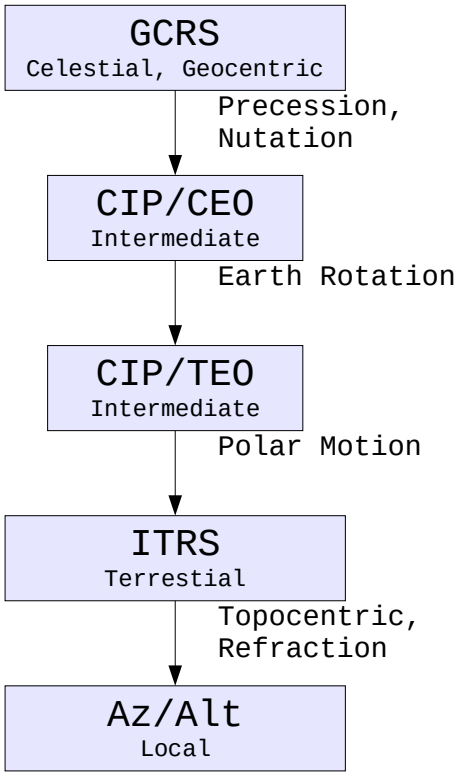

Figure 2: Earth Orientation Coordinate Frames

7.5.1.1 Aberration

The following function calculates the `apparent` position of a star, given its `actual` position and the velocity vector of the observer, represented as a speed and a direction:

```
psSphere *psAberration(psSphere *apparent, const psSphere *actual, const psSphere *direction, double speed)
```

The `actual` and `apparent` positions are represented as `psSphere` entries, as is the `direction` of motion. The speed in that direction is given in units of the speed of light. If the value of `apparent` is NULL, a new `psSphere` is allocated, otherwise the point to `apparent` is used for the result.

7.5.1.2 Gravitational Deflection

The following function calculates the `apparent` position of a star, given its `actual` position and the position of the sun:

```
psSphere *psGravityDeflection(psSphere *apparent, psSphere *actual, psSphere *sun);
```

The `actual` and `apparent` positions are represented as `psSphere` entries, as is position of the sun. If the value of `apparent` is NULL, a new `psSphere` is allocated, otherwise the point to `apparent` is used for the result.

7.5.1.3 Parallax

The parallax factor is not critical to the EOC calculations, and we don't have a formula handy for it at the moment, so do not code. (TBD)

```
double psEOC_ParallaxFactor(const psSphere *coords, const psTime *time);
```

Calculate the parallax factor for the given position and time.

7.5.2 Transformation from GCRS to ITRS

The following functions calculate the components, X, Y, and s, representing the location of the earth's pole at any moment, or they determine the velocity of the pole X', Y', s'. We use the following structure to carry the polar coordinate information. This representation may be converted to a rotation between the frames.

```
typedef struct {
  double x;
  double y;
  double s;
} psEarthPole;
```

The constructor is:

```
psEarthPole *psEarthPoleAlloc(void);
```

7.5.2.1 Precession/Nutation

The following routine calculates the components of the rotation between the CEO and GCRS frames, X, Y, and s, using to the IAU2000A precession & nutation model:

```
psEarthPole *psEOC_PrecessionModel(const psTime *time)
```

The input to this function is the desired `time`, which may be represented in any format other than UT1. This routine must give results identical to the IERS XYS2000A subroutine within the limits of machine accuracy.

The following function provides interpolated corrections to the X and Y components of the polar coordinates from the tables provided by the IERS, just as it does for UT1 and polar motion.

```
psEarthPole *psEOC_PrecessionCorr(const psTime *time, psTimeBulletin bulletin);
```

The polar correction is applied to the X and Y elements of the rotation to provide higher accuracy. The spherical rotation term is generated by providing the polar coordinate to the following function:

```
psSphereRot *psSphereRot_CEOtoGCRS(const psEarthPole *pole)
```

This function constructs the rotation element as described in the ADD (The resulting `psSphereRot` may be used to determine the rotation from CIP/CEO to GCRS. This function must give results identical to the IERS BPN2000, within the limits of machine accuracy.

7.5.2.2 Earth Rotation

The following routine calculates the rotation of the Earth about the CIP:

```
psSphereRot *psSphereRot_TEOtoCEO(const psTime *time, psEarthPole *tidalCorr)
```

The IERS code to create the comparable rotation is embedded in T2C2000, with the Earth Rotation Angle calculated by ERA2000. The tidal correction, `tidalCorr` from `psEOC_PolarTideCorr` is used to correct UT1.

7.5.2.3 Polar Motion

The following function provides interpolated values of the polar motion components, x_p and y_p, extracted from the IERS tables.

```
psEarthPole *psEOC_GetPolarMotion(const psTime *time, psTimeBulletin bulletin);
```

The following function provides tidal corrections to the polar motion components, x_p and y_p, using the Ray model of Simon et al (see ADD). It also provides a time correction to UT1 for `psSphereRot_TEOtoCEO` that we will, for convenience, place in the `s` component of the output `psEarthPole`.

```
psEarthPole *psEOC_PolarTideCorr(const psTime *time);
```

The following function provides the additional corrections due to nutation terms with periods less than or equal to two days, as well as the correction to the s' component of the polar motion:

```
psEarthPole *psEOC_NutationCorr(psTime *time);
```

The following function converts the polar motion corrections to a spherical rotation using the prescription in the ADD:

```
psSphereRot *psSphereRot_ITRStoTEO(const psEarthPole *motion);
```

This function should give identical results to the IERS POM2000 subroutine.

7.5.3 Earth Orientation Wrappers

The following function generates the complete spherical rotation to account for precession between two times. If `NULL` is provided for either time, it is assumed to have the reference equinox value of J2000.

```
psSphereRot *psSpherePrecess(const psTime *fromTime, const psTime *toTime, psPrecessMethod mode);
```

The mode argument is used to specify the level of detail used in the calculation.

```
typedef enum {
  PS_PRECESS_ROUGH,
  PS_PRECESS_COMPLETE_A,
  PS_PRECESS_COMPLETE_B,
  PS_PRECESS_IAU2000A
} psPrecessMethod;
```

`PS_PRECESS_ROUGH` indicates that an approximate precession rotation is determined from a cubic polynomial in the time difference (ADD 3.4.3). `PS_PRECESS_IAU2000A` indicates that the precession rotation is determined from differencing the two rotations obtained from applying the IAU 2000A model (`psEOC_PrecessionModel`, followed by `psSphereRot_CEOtoGCRS`) at each epoch. `PS_PRECESS_COMPLETE_A` and `PS_PRECESS_COMPLETE_B` indicates that the precession rotation is the same as for `PS_PRECESS_IAU2000A`, except that the earth pole correction published by the IERS is included (using the appropriate bulletin in `psEOC_PrecessionCorr` — `PS_IERS_A` for `PS_PRECESS_COMPLETE_A`, and `PS_IERS_B` for `PS_PRECESS_COMPLETE_B`).

7.6 Atmospheric Effects

The ATM effects components should be deferred until we clean up the refraction definitions (TBD)

A-priori astrometric transformations between the telescope orientation (Alt/Az) and the predicted stellar coordinates above the atmosphere (DEC/HA) requires several pieces of information describing the current environmental conditions. These quantities are consistent across an image, and may vary only slowly with time. Pre-computing these quantities for exposures means that subsequent transformations are faster. The structure below carries the environmental data of interest. For historical reasons, this structure is known colloquially as “the Grommit”.

this structure needs to be modified to correspond to what we actually need to carry around for the atmosphere functions (TBD)

provide a single Grommit to carry around all EOC + ATM pre-calculated entries and a separate structure for ATM effect? (TBD)

```
 %%% Changed to verbatim to remove from api-delta
typedef struct {
    const double latitude;              ///< geodetic latitude (radians)
    const double longitude;             ///< longitude + ... (radians)
    const double height;                ///< height (HM)
    const double abberationMag;         ///< magnitude of diurnal aberration vector
    const double temperature;           ///< ambient temperature (TDK)
    const double pressure;              ///< pressure (PMB)
    const double humidity;              ///< relative humidity (RH)
    const double wavelength;            ///< wavelength (WL)
    const double lapseRate;             ///< lapse rate (TLR)
    const double refractA, refractB;    ///< refraction constants A and B (radians)
    const double siderealTime;          ///< local apparent sidereal time (radians)
} psGrommit;
```

The `psGrommit` is calculated from telescope information for the particular exposure, `exp`:

```
psGrommit *psGrommitAlloc(const psExposure *exp);
```

these functions probably need to take the ATM structure (TBD)

We require additional functions to perform general functions which will be useful for astrometry. Given coordinates on the sky, we need to get the airmass, the parallactic angle, and an estimate of the atmospheric refraction.

```
float psGetAirmass(const psSphere *coord, psTime *lst, float altitude);
```

which returns the airmass for a given position and local sidereal time (`lst`).

```
float psGetParallactic(const psSphere *coord, double siderealTime);
```

which returns the parallactic angle for a given position and sidereal time.

```
float psGetRefraction(float colour,           ///< Colour of object
                      psPhotSystem colorPlus,  ///< Colour reference
                      psPhotSystem colorMinus, ///< Colour reference
                      const psExposure *exp);  ///< Telescope pointing information
```

which provides an estimate of the atmospheric refraction, along the parallactic angle.

7.7 Projections

We require functions to convert between spherical and linear coordinate systems based on a variety of projections. The required projections include:

- TAN
- SIN
- AIT
- PAR

We specify the following structure `psProjection` to define the parameters of the projection:

```
typedef struct {
    double R;                           ///< coordinates of projection center
    double D;                           ///< coordinates of projection center
    double Xs;                          ///< plate-scale in X and Y directions
    double Ys;                          ///< plate-scale in X and Y directions
    psProjectionType type;              ///< projection type
} psProjection;
```

The projection type is defined by the following enumerated type `psProjectionType`:

```
typedef enum {                          ///< type of val is:
    PS_PROJ_TAN,                        ///< Tangent projection
    PS_PROJ_SIN,                        ///< Sine projection
    PS_PROJ_AIT,                        ///< Aitoff projection
    PS_PROJ_PAR,                        ///< Par projection
    PS_PROJ_NTYPE                       ///< Number of types; must be last
} psProjectionType;
```

The constructor is straight-forward:

```
psProjection *psProjectionAlloc(double R, double D, double Xs, double Ys, psProjectionType type);
```

where the units of `R` and `D` are radians, while `Xs` and `Ys` are in radians per pixel.

The following functions will project and deproject (respectively) spherical coordinates:

```
psPlane  *psProject(const psSphere *coord, const psProjection *projection);
psSphere *psDeproject(const psPlane *coord, const psProjection *projection);
```

7.8 Astronomical objects

These functions are all of low priority, have not yet been defined in detail, and hence are to be deferred. (TBD)

7.8.1 Positions of Major SS Objects

We may require the ability to calculate the position of major Solar System objects, as well as Lunar phase.

```
 %%% XXX: This is set to 'verbatim' instead of 'prototype'
psSphere *psSunGetPos(psTime *time);
psTime *psSunGetRise (psTime *twi15, psTime *twi18, const psTime *time);
psTime *psSunGetSet (psTime *twi15, psTime *twi18, const psTime *time);

psSphere *psMoonGetPos(psTime time, psSphere location);
psTime *psMoonGetRise (psTime *twi15, psTime *twi18, psTime *time);
psTime *psMoonGetSet (psTime *twi15, psTime *twi18, psTime *time);
float psGetMoonPhase(psTime time);

psSphere *psPlanetGetPos(psTime time, psSphere location);
psTime *psPlanetGetRise (psTime *twi15, psTime *twi18, psTime *time);
psTime *psPlanetGetSet (psTime *twi15, psTime *twi18, psTime *time);
```

A Configuration File Test Inputs

Here are a series of test inputs for the Configuration File syntax defined in §3.6.4.

A.1 Complete Examples

SDRS example

Pass. Test with and without overwrite.

```
Double     F64     1.23456789      # This is a comment
Float    F32 0.98765#This is a comment too
String  STR This is the string that forms the value #comment

 # This is a comment line and is to be ignored
boolean     BOOL    T # The value of 'boolean' is 'true'

@primes U8  2,3 5 7,11,13 17 #   These are prime numbers

comment MULTI # The rest of this line is ignored, but 'comment' is set to be non-unique
comment STR This
comment STR     is
comment STR       a
comment STR        non-unique
comment STR                 key
Float F64 1.23456 # This generates a warning, and, if 'overwrite' is 'false', is ignored
```

Gene's example

Pass. Overwrite warning.

```
# these are examples of camera definition variables:
# skyprobe
NCELL       S32    1
NCHIP       S32    1
#                  FILENAME    EXTNAME  REGION      CHIP      BIASSEC
CELL.00     STR    %f00.%x     PHU      [0,0:0,0]   CHIP.00   [0,0:0,0]

# megacam-raw
NCELL       S32    72
NCHIP       S32    36
#                  FILENAME    EXTNAME  DATASEC     CHIP      BIASSEC
CELL.00     STR    %f.%x       AMP00    [0,0:0,0]   CHIP.00   BIASSEC
CELL.01     STR    %f.%x       AMP01    [0,0:0,0]   CHIP.00   [2100,2110:0,4096]
CELL.02     STR    %f.%x       AMP02    [0,0:0,0]   CHIP.01   [0,0:0,0]
CELL.03     STR    %f.%x       AMP03    [0,0:0,0]   CHIP.01   [0,0:0,0]

# megacam-splice
NCELL       S32    72
NCHIP       S32    36
#                  FILENAME    EXTNAME  REGION      CHIP      BIASSEC   TRIMSEC
CELL.00     STR    %f.%x       CCD00    ASEC-00     CHIP.00   BSEC-00   DSEC-00
CELL.01     STR    %f.%x       CCD00    ASEC-01     CHIP.00   BSEC-01   DSEC-01
CELL.02     STR    %f.%x       CCD01    ASEC-00     CHIP.01   BSEC-00   DSEC-00
CELL.03     STR    %f.%x       CCD01    ASEC-01     CHIP.01   BSEC-01   DSEC-01

# cfh12k-split
NCELL       S32    12
```

```
NCHIP       S32    12
#                  FILENAME    EXTNAME  REGION      CHIP      BIASSEC
CELL.00     STR    %f/%f00.%x  PHU      [0,0:0,0]   CHIP.00   [0,0:0,0]
CELL.01     STR    %f/%f01.%x  PHU      [0,0:0,0]   CHIP.01   [0,0:0,0]
CELL.02     STR    %f/%f02.%x  PHU      [0,0:0,0]   CHIP.02   [0,0:0,0]

# cfh12k-mef
NCELL       S32    12
NCHIP       S32    12
#                  FILENAME    EXTNAME  REGION      CHIP      BIASSEC
CELL.00     STR    %f.%x       CHIP00   [0,0:0,0]   CHIP.00   [0,0:0,0]
CELL.01     STR    %f.%x       CHIP01   [0,0:0,0]   CHIP.01   [0,0:0,0]
CELL.02     STR    %f.%x       CHIP02   [0,0:0,0]   CHIP.02   [0,0:0,0]

#- REGION can be defined by a header keyword in IRAF format or by a explicit IRAF format
#- what is default for NAXIS1,2 for IRAF format?
#- Nreadout is always NAXIS3?

# recipe file:
# this makes the assumption that, for a given camera, all chips &
# cells have the same recipe.  this is probably a good start, but may
# not cut it in general. eg, it is already clear that for

# recipe file must be a function of time and camera.
#

# BIAS:
BIAS.IMAGE                 STR    NONE
BIAS.IMAGE      STR    FILE:bias.fits
BIAS.IMAGE      STR    DB:BEST
BIAS.IMAGE      STR    DB:CLOSE

BIAS.OVERSCAN     STR    HEADER:BIASSEC
BIAS.OVERSCAN     STR    RECIPE:[0,0:0,0]
BIAS.OVERSCAN     STR    NONE

BIAS.OVERSCAN.STATS     STR    MEDIAN
BIAS.OVERSCAN.STATS     STR    MEAN

BIAS.OVERSCAN.FIT          STR    SPLINE
BIAS.OVERSCAN.FIT.NPTS     S32    5

BIAS.OVERSCAN.FIT          STR    POLYNOMIAL
BIAS.OVERSCAN.FIT.ORDER    S32    3
BIAS.OVERSCAN.FIT.NBIN     S32    5
```

A.2 METADATA

SDRS example

Pass.

```
CELL      METADATA
 EXTNAME   STR   CCD00
 BIASSEC   STR   BSEC-00
 CHIP      STR   CHIP.00
 NCELL     S32   24
END
```

nested metadata

Pass.

```
CELL      METADATA
    FOO METADATA
        BAR     STR BAZ
        PING    STR PONG
    END

    EXTNAME   STR   CCD00
    BIASSEC   STR   BSEC-00
    CHIP      STR   CHIP.00
    NCELL     S32   24
END
```

deeply nested metadata

Pass.

```
FOO1 METADATA
    FOO2 METADATA
        FOO3 METADATA
            FOO4 METADATA
                FOO5 METADATA
                    FOO6 METADATA
                        BAR     STR BAZ
                        PING    STR PONG
                    END
                    BAR     STR BAZ
                    PING    STR PONG
                END
                BAR     STR BAZ
                PING    STR PONG
            END
            BAR     STR BAZ
            PING    STR PONG
        END
        BAR     STR BAZ
        PING    STR PONG
    END
    BAR     STR BAZ
    PING    STR PONG
END
```

deeply nested metadata

Pass.

```
FOO1 METADATA
    BAR     STR BAZ
    PING    STR PONG
    FOO2 METADATA
        BAR     STR BAZ
        PING    STR PONG
        FOO3 METADATA
            BAR     STR BAZ
            PING    STR PONG
```

```
            FOO4 METADATA
                BAR     STR BAZ
                PING    STR PONG
                FOO5 METADATA
                    BAR     STR BAZ
                    PING    STR PONG
                    FOO6 METADATA
                        BAR     STR BAZ
                        PING    STR PONG
                    END
                END
            END
        END
    END
END
```

deeply nested metadata

Pass.

```
FOO1 METADATA
    BAR     STR BAZ
    FOO2 METADATA
        BAR     STR BAZ
        FOO3 METADATA
            BAR     STR BAZ
            FOO4 METADATA
                BAR     STR BAZ
                FOO5 METADATA
                    BAR     STR BAZ
                    FOO6 METADATA
                        BAR     STR BAZ
                        PING    STR PONG
                    END
                    PING    STR PONG
                END
                PING    STR PONG
            END
            PING    STR PONG
        END
        PING    STR PONG
    END
    PING    STR PONG
END
```

two metadata at the same depth

Pass.

```
FOO1 METADATA
    FOO2 METADATA
        BAR     STR BAZ
        PING    STR PONG
    END
    FOO3 METADATA
        BAR     STR BAZ
        PING    STR PONG
    END
END
```

A.3 TYPE

basic TYPE

Pass.

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP
CELL.00   CELL   CCD00     BSEC-00  CHIP.00
CELL.01   CELL   CCD01     BSEC-01  CHIP.00
```

TYPE with comments

Pass.

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP    # comment
CELL.00   CELL   CCD00     BSEC-00  CHIP.00 # foo
CELL.01   CELL   CCD01     BSEC-01  CHIP.00 #
```

TYPE not visible in lower scopes

Pass.

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP
CELL.00   CELL   CCD00     BSEC-00  CHIP.00
CELL.01   CELL   CCD01     BSEC-01  CHIP.00
FOO METADATA
    TYPE      CELL   EXTNAME   BIASSEC
    CELL.00   CELL   CCD00     BSEC-00
    CELL.01   CELL   CCD01     BSEC-01
    FOO METADATA
        TYPE      CELL   EXTNAME
        CELL.00   CELL   CCD00
        CELL.01   CELL   CCD01
    END
END
```

TYPE not in scope

Fail.

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP
FOO METADATA
    CELL.00   CELL   CCD00     BSEC-00  CHIP.00
END
```

TYPE not in scope

Fail.

```
FOO METADATA
    TYPE      CELL   EXTNAME   BIASSEC  CHIP
END
CELL.00   CELL   CCD00     BSEC-00  CHIP.00
```

TYPE not in scope

Fail.

```
FOO METADATA
    TYPE      CELL   EXTNAME   BIASSEC  CHIP
END
BAR METADATA
    CELL.00   CELL   CCD00     BSEC-00  CHIP.00
END
```

TYPE with missing parameters

Fail.

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP
CELL.00   CELL   CCD00     BSEC-00
CELL.01   CELL   CCD01     BSEC-01  CHIP.00
```

TYPE redefinition

Fail.

```
TYPE      CELL   EXTNAME   BIASSEC  CHIP
TYPE      CELL   EXTNAME   BIASSEC  CHIP
CELL.00   CELL   CCD00     BSEC-00  CHIP.00
CELL.01   CELL   CCD01     BSEC-01  CHIP.00
```

missing TYPE declaration

Fail.

```
CELL.00   CELL   CCD00     BSEC-00  CHIP.00
CELL.01   CELL   CCD01     BSEC-01  CHIP.00
```

A.4 Time

basic IS08601

Pass.

```
recently    UTC     2005-03-18T16:05:00Z
recently    UT1     2005-03-18T16:05:00Z
recently    TAI     2005-03-18T16:05:00Z
recently    TT      2005-03-18T16:05:00Z
```

ISO8601 with comments

Pass.

```
recently    UTC     2005-03-18T16:05:00Z    # foo
recently    UT1     2005-03-18T16:05:00Z    # bar
recently    TAI     2005-03-18T16:05:00Z    # baz
recently    TT      2005-03-18T16:05:00Z    #
```

bad format

Fail.

```
broken      UTC     2005-03-18T16:05:00
```

bad format

Fail.

```
broken      UT1     2005-03-18T16:05:00
```

bad format

Fail.

```
broken      TAI     2005-03-18T16:05:00
```

bad format

Fail.

```
broken      TT      2005-03-18T16:05:00
```

A.5 MULTI

basic MULTI

```
foo MULTI
foo S8      -1
foo STR     bar baz
foo BOOL    T
```

MULTI with comments

Pass.

```
foo MULTI               # foo
foo S8      -1          # bar
foo STR     bar baz     # baz
foo BOOL    T           #
```

MULTI not visible in lower scopes

Pass.

```
foo MULTI
foo S8      -1
foo STR     bar baz
foo BOOL    T
bar METADATA
    foo MULTI
    foo S8      -1
    foo STR     bar baz
    foo BOOL    T
END
```

MULTI METADATA

Pass.

```
foo MULTI
foo METADATA
    bar BOOL    T
END
foo METADATA
    bar BOOL    T
END
```

MULTI METADATA structure, not declared

Pass. Overwrite warning.

```
foo METADATA
    bar BOOL    T
END
foo METADATA
    bar BOOL    T
END
```

MULTI TYPE

Pass.

```
foo MULTI
TYPE bar a b c
foo  bar x y z
foo  bar x y z
```

MULTI TYPE

Pass.

```
TYPE bar a b c
foo MULTI
foo  bar x y z
foo  bar x y z
```

MULTI TYPE, not declared

Pass. Overwrite warning.

```
TYPE bar a b c
foo  bar x y z
foo  bar x y z
```

MULTI not in scope

Pass. Overwrite warning.

```
foo MULTI
bar METADATA
    foo S8      -1
    foo STR     bar baz
END
```

MULTI not in scope

Pass. Overwrite warning.

```
bar METADATA
    foo MULTI
END
foo S8      -1
foo STR     bar baz
```

MULTI not in scope

Pass. Overwrite warning.

```
bar METADATA
    foo MULTI
END
baz METADATA
    foo S8      -1
END
```

MULTI redeclaration

Fail.

```
foo MULTI
foo MULTI
foo S8      -1
```

missing MULTI declaration

Pass. Overwrite warning.

```
foo S8      -1
foo STR     bar baz
```

B Dates & Times Test Inputs

Here are a series of test inputs for the `psTime`(§7.1) ISO8601 parsing and formatting functions. These tests will also validate the behavior of `psTime`'s conversion algorithms.

B.1 Equivalent Dates/Times

```
1999-01-01T00:00:29.000Z TAI
1998-12-31T23:59:58.000Z UTC
1998-12-31T23:59:57.716Z UT1
1999-01-01T00:01:01.184Z TT

1999-01-01T00:00:29.500Z TAI
1998-12-31T23:59:58.500Z UTC
1998-12-31T23:59:58.216Z UT1
1999-01-01T00:01:01.684Z TT

1999-01-01T00:00:30.000Z TAI
1998-12-31T23:59:59.000Z UTC
1998-12-31T23:59:58.716Z UT1
1999-01-01T00:01:02.184Z TT

1999-01-01T00:00:30.500Z TAI
1998-12-31T23:59:59.500Z UTC
1998-12-31T23:59:59.216Z UT1
1999-01-01T00:01:02.684Z TT

1999-01-01T00:00:31.000Z TAI
1998-12-31T23:59:60.000Z UTC
1998-12-31T23:59:59.716Z UT1
1999-01-01T00:01:03.184Z TT

1999-01-01T00:00:31.500Z TAI
1998-12-31T23:59:60.500Z UTC
1999-01-01T00:00:00.216Z UT1
1999-01-01T00:01:03.684Z TT

1999-01-01T00:00:32.000Z TAI
1999-01-01T00:00:00.000Z UTC
1999-01-01T00:00:00.716Z UT1
1999-01-01T00:01:04.184Z TT
```

```
1999-01-01T00:00:32.500Z TAI
1999-01-01T00:00:00.500Z UTC
1999-01-01T00:00:01.216Z UT1
1999-01-01T00:01:04.684Z TT
```

C Revision Change Log

C.1 Changes from version 00 to version 01

- cosmetic re-organizations
- specified data types valid for psVector and psImage functions
- discussion of external libraries
- minor discussion of threads
- memory callback routine names changed from ...CB to ...Callback
- added discussion of freeing/dereferencing components of structures
- renamed psPrintTraceLevels -¿ psTracePrintLevels, added prototype
- added psTraceSetDestination
- changed psVLogMsg to psLogMsgV
- changed psSetLogDestination to psLogSetDestination
- changed psLogSetDestination destination argument to type char * extended output target concept
- changed psSetLogFormat to psLogSetFormat
- extended the concept of psError to define an error stack
- added psErrorGet, psErrorLast, psErrorClear
- defined psErr structure
- added psErrorStackPrint & psErrorStackPrintV
- added psErrorCodeString
- added psErrorRegister
- defined psErrorDescription
- dropped psStringCopy & psStringNCopy
- changed naming scheme for psElemType to PS_TYPE_F32 format
- dropped psFLoatArray, psIntArray, psDoubleArray, psComplexArray, psVoidPtrArray, and associated functions
- defined psVector, psVectorAlloc, psVectorRealloc, psVectorFree

- changed naming scheme for psImage union (rows.rows_f32 -¿ data.F32)
- added psElemType entry to psImageAlloc and psVectorAlloc
- added 'which' argument to psDlistSetIterator, psDlistGetNext, psDlistGetPrev
- added psDlistSort function
- changed 'bitmask' to 'bitset'
- added psBitsetNot function
- changed output of psSort to psVector
- changed psArrayStats to psVectorStats
- dropped robustMeanNvalues, etc from psStats
- added robustN50, robustNfit, binsize to psStats
- reduced available options values (psStatsOptions)
- simplified psHistogram, psHistogramAllocGeneric
- changed psGetArrayHistogram to psHistogramVector
- moved Matrix section after Images
- moved FFT section after Images
- replaced psMatrixOp with psMatrixMultiply
- changed FFT function names to have the forms psVectorFFT and psImageFFT
- replaced the forward and reverse version with an argument to ps...FFT
- added ps...Real, ps...Imaginary, ps...Complex, ps...Conjugate
- changed psEvalPolynomial1D to psPolynomial1DEval, (and equivalents)
- added 'normal' argument to psGaussian
- added psGaussianDev
- modified argument lists of psMinimize and psMinimizeChi2
- changed psGetVectorPolynomial to psVectorFitPolynomial1D
- dropped psImageFreeChildren
- modified union naming scheme in p_psScalar to data.S32 format
- added Dates and Times functions:
- modified union naming scheme in psMetadataType to data.S32 format
- modified psMetadataType to use S32, etc types

- changed arguments of psMetadataItemAlloc to use stdargs to get data value
- added psMetadataItemAllocV
- modified psMetadataAppend arglist to match psMetadataItemAlloc
- changed psImageReadHeader to psMetadataReadHeader (& Fread)
- moved psMetadataReadHeader to metadata section
- split psCoord into psPlane and psSphere
- renamed psCoordXform to psPlaneTransform
- renamed psDistortion to psPlaneDistortion
- defined psSphereTransform, ..Alloc, ..Free
- changed psCoordTransform to psSphereTransformApply
- delete psCoordinatesItoE & equivalents
- added psShereTransformItoE & equivalents
- defined psProjection, psProjectionType
- modifed args to psProject, psDeproject
- changed psGetOffset, psApplyOffset to psSphereGetOffset, psSphereSetOffset
- dropped overscan entry from psReadout
- dropped grommit from psExposure
- added psGrommitAlloc, psGrommitFree
- changed args to psCoordsSkyToCell & psCoordsSkyToCellQD
- modified argument lists for most psCoord conversion functions
- renamed psGetSun, psGetMoon functions to psSunGet, psMoonGet
- added ps...GetRise, ps...GetSet, psNightLength
- added psTimeToLunation, psLunationToTime
- moved some naming issues out of this document, to IPP SRS
- cleaned metadata discussion
- change psLog date format to use hyphens
- change Configuration File Grammar to allow periods in names and strings that begin with non-word characters

C.2 Changes from Revision 01 (19 May 2004) to 02 (22 June 2004)

Changes consisted of incorporating feedback from Bugzilla problem reports (up to bug number 64).

C.3 Changes from Revision 02 (22 June 2004) to 06 (19 August 2004)

- `libTAI` no longer used, since it does not perform as desired.
- Addition of `previousBlock`, `nextBlock`, `freeFcn`, `userMemorySize`, `refCounterMutex` to `psMemBlock`.
- Added behavior of memory management when `PS_MEM_DEBUG` is defined at compile time.
- `psMemExhaustedSetCallback` → `psMemExhaustedCallbackSet`.
- Added `p_psSetFreeFcn` and `p_psGetFreeFcn`.
- `psMemFreeIDSet`, `psMemAllocateIDSet` → `psMemFreeCallbackIDSet`, `psMemAllocateCallbackIDSet`.
- `psMemCheckCorruption` now takes a `bool` instead of `int`.
- Specification that destructors are private functions which are called by `psFree`.
- `psErrorRegisterSet` $rightarrow$ `psErrorRegister`.
- Added a type, `PS_TYPE_C64`.
- Added section on "Simple Scalars" for math operations, including functions `psScalarAlloc` and `psScalarCopy`.
- `psVector` no longer carries a `void *` type, but now has a `psC64` type. The `void *` carrier type is `psArray`.
- `psImage` supporting functions are valid for types `psS8, psS16, psU8, psU16, psF32, psF64, psC32, psC64`.
- Section on "Simple Arrays" added — an ordered collection of unspecified data elements.
- `psDList` renamed `psList`.
- In `psList`, `n` changed to `size`.
- `psListAdd` and `psListRemove` now return a `bool`.
- Retrieving data from the list means that the reference counter is incremented.
- `psListFree` frees the list and calls `psFree` on all the data on the list.
- `psListSort` prototype updated.
- `psHashInsert` renamed `psHashAdd`, now returns a `bool` and frees existing data for the given key.
- `psHashRemove` now returns a `bool`.
- `psSort` renamed `psVectorSort`, and specified for type `psS8`, instead of `psU8`.
- `psSortIndex` renamed `psVectorSortIndex`.
- `psVectorStats` parameters reordered.
- `psStats` does not revert to robust statistics for large vectors; `PS_STAT_ROBUST_FOR_SAMPLE` dropped.

- `psHistogram.uniform` changed to type `bool`.
- `psHistogramVector` renamed `psVectorHistogram`, and takes a `mask` and `maskVal`.
- `psPolynomial` now incorporates both general polynomials and Chebyshev polynomials, with the option being specified by an enumerated type in the structure, which is also passed to the constructor, `psPolynomialAlloc`. This will necessitate changes to the evaluators as well.
- Vector versions of polynomial evaluators, e.g. `psDPolynomial2DEvalVector` defined.
- Splines added (`psSpline1D`), with corresponding allocators, single and vector evaluators, and fit to a vector.
- `psGaussian` takes `bool normal`, instead of `int normal`.
- Specified minimization routines in much more detail. Now have both LM and Powell minimizers specified, with a χ^2 minimizer using the Powell minimizer, and a couple of functions to use to minimize χ^2 with the LM minimizer.
- Function `psImageSubsection` added.
- Behavior of `psImageCopy` specified in the event `out == NULL`.
- Function `psImageTrim` added.
- Defined `psImageCutDirection`, and added one to `psImageSlice`.
- `mask` and `maskVal` added to `psImageCut`, along with pointer to the algorithm in the ADD.
- `mask` and `maskVal` added to `psImageRadialCut`, along with pointer to the algorithm in the ADD..
- `mask` and `maskVal` added to `psImageRebin`, and specified in more detail.
- Added function `psImageResample`, along with `psImageResampleMode`.
- `psImageRotate` has new `float exposed` parameter.
- `psImageGetStats` renamed `psImageStats`, and must be defined for `psS8` (not `psU8`).
- `psImageHistogram` now takes a `mask` and `maskVal`.
- `psImageHistogram`, `psImageFitPolynomial`, `psImageEvalPolynomial` must be defined for `psS8` (not `psU8`).
- Added function `psImagePixelInterpolate`.
- Behavior of `psImageReadSection, psImageWriteSection` specified in more detail.
- `psImageClip` parameters now specified to be `double`, and behavior specified in more detail.
- Function `psImageClipComplexRegion` added.
- `psImageClipNaN` behavior specified for complex images.
- `psBinaryOp` and `psUnaryOp` described in more detail.
- `psVectorTranspose` removed.

- Added section on convolution, including defining `psKernel`, with corresponding constructor and generation function.
- Section on times (`psTime` completely reworked. In particular, `libTAI` is no longer used.
- `psMetadataAppend` and `psMetadataAppendItem` are now `psMetadataAdd` and `psMetadataAddItem`, returning `bool`s. Each is specified in a bit more detail.
- `psMetadataRemove` takes a `where` parameter, and is specified in a bit more detail.
- Added function `psMetadataGet`.
- Iterating on the metadata described in more detail.
- `psMetadataItemPrint` changed.
- In distortions, the third and fourth parameters consistently referred to now in the order color and then magnitude.
- Pre-defined spherical transforms renamed to longer, but more meaningful names.
- GLS projection dropped.
- Offsets specified in more detail, with different modes (`psSphereOffsetMode`) and units (`psSphereOffsetUnit`) instead of a single `type`.
- `out` parameter in `psCellInFPA, psChipInFPA, psCellInChip` removed.
- `color, mag` parameters added to functions that transform between FPA and tangent plane (`psCoordFPAToTP, psCoordCellToSky, psCoordTPToFPA, psCoordSkyToCell`).
- Specified constructors for the astronomy images and astrometry structures.
- `psExposure` contains a `psTime *time` instead of `mjd`.
- `psFPA` contains a `psGrommit` composed from the relevant `psExposure` for assistance in astrometric transforms.
- Added structure `psObservatory`, which contains observatory information. `psExposure` contains a `psObservatory`.
- `wavelength` added to `psExposure`.
- `psCell.cellToSky` renamed to `psCell.toTP` and now only goes to the tangent plane, from which the `psGrommit` is used to get the coordinates to the sky.
- Added description of `psMemory` functions `p_psSetFreeFcn`, `p_psGetFreeFcn`
- Modified definition of `psImagePixelInterpolate`.
- Renamed `psImageResampleMode` mode to `psImageInterpolateMode`.
- Added `psImageInterpolateMode` mode to `psImageRotate`, `psImageShift`.
- Changed input type of `exposed` pixels to `psC64`.
- Dropped maintaince of original type for `psImageCopy`.

- Added output `psVector *coords` to `psImageSlice`.
- Dropped `psMetadataFlags` from `psMetadataItem`.
- Cleaned `psMetadata` discussion to reflect new handling of non-unique keys.
- Clarified sequence of entries in `psMetadataItemAlloc`.
- Dropped metadata naming convention reference (should be part of IPP docs, not PSLib)
- `Nchildren` → `nChildren` in `psImage`.
- Changed names of `p_psSetFreeFcn`, `p_psGetFreeFcn` to `p_psMemSetDeallocator`, `p_psMemGetDeallocator`.
- Added `psFreeFnc` parameter to `p_psMemSetDeallocator`.
- `long psMemGetId(void)` to `psMemoryId psMemGetId(void)`.

C.4 Changes from Revision 06 (19 August 2004) to Revision 07 (7 September 2004)

- Removed `psTimeFromLST`, changed parameters and output for `psTimeToLST`.
- Added function `psTimeLeapSeconds` to calculate number of leap seconds between two dates, required for time arithmetic.
- Changed `psImageCut` to do a slice through the data in an arbitrary direction, instead of merging the data along a particular dimension.
- added `psVector *coords` to the API for `psImageCut`.
- Added `psLibVersion` (bug 156).
- Added specification of iterator to `psMetadataSetIterator`.
- Replaced `which` and `where` in `psMetadata` and `psList` functions with `iterator` and `location`, respectively, in order to reduce confusion.
- Changed the order of some `psList` function arguments to match that of `psMetadata`.
- `psLogSetDestination` return type changed to `bool`.
- `psImage.nChildren` and `.children` changed to `psArray *children`.
- Added extra input parameters to `psKernelGenerate`: `psVector *tShifts` and `bool relative`.
- Synched `psImageSubset`, `psImageTrim`, `psImageReadSection` and `psImageSlice` to use a consistent region specification: `x0,y0,x1,y1`, where `x1` and `y1` are exclusive, and may be negative.
- Added `psTime` arguments to `psSphereTransformICRSToEcliptic` and `psSphereTransformEclipticToICRS`.
- Added `bool` type to `psMetadata`.
- TBD-ed the Astronomical Objects section and the photometry section.

- Added psLibInit.
- Updated psReadout,Cell,Chip,FPA to synchronise with MHPCC, in particular use of psArray.
- psImageTrim is to free children.
- Specified the operators for psBinaryOp and psUnaryOp.
- Updated types for psVectorStats, psImageReadSection and psImageWriteSection.
- Added constructors for psPlaneTransform and psPlaneDistort.
- Updated description of behavior of psFree (doesn't barf if refCounter is not 1.
- Clarified behavior of psLogSetDestination (no need to specify a protocol for stderr,stdout,none).
- Leading dot in facility name for psTrace is optional.
- Specification of configuration file format and parse function, psMetadataParseConfig.
- Added new function psSpherePrecess.

C.5 Changes from Revision 07 (7 September 2004) to Revision 08 (12 October 2004)

- Changed format of psLogMsg, following bug 189: format now has multiple lines.
- Added configuration file grammar.
- Added explanation of psArray *coords in psMinimizeLMFunc, and changed psMatrix to psImage in response to bug 191.
- Added psTimeTable.
- psLibInit altered to load psTime configuration file.
- Reworked psList (including addition of new functions) to support multiple iterators.
- Specified return value for psImageOverlaySection is the number of pixels overlaid.
- Added persistent to psMemBlock, along with additional specification of behaviour of psMemCheckLeaks.
- Removed psListRemoveNext and psListRemoveAfter. Removing an item pointed to by an iterator doesn't cause an error.
- Added psRectangle
- Added PS_TYPE_BOOL to psElemType.
- Dropped PS_TYPE_OTHER
- Replaced PS_META_IMG with PS_META_MATH
- Replaced PS_META_ITEM_SET with PS_META_LIST
- Added psElemType to psMetadataItem to specify primitive data types.

- Cleaned up the entries in `psMetadataType`.
- Moved the `psList` entry (item set, now list) to the union.
- Added a `psMetadata` entry to the union.
- Added typing contruct and hierarchy to `psMetadataParseConfig`
- Added FITS I/O section
- Moved `psImageReadSection` and `psImageWriteSection` to FITS I/O Functions section and modified names.
- Moved `psMetadataReadHeader` and `psMetadataFReadHeader` to FITS I/O Functions section and modified names.
- Fixed typo: `p_psScalar` to `psScalar`
- Renamed `psMetadata.data.void` to `psMetadata.data.data` to be consistent with `psList`, `psHash`
- Require comment mark in 'multiple' metadata config entry for comments
- Changed return values for `psMetadataParseConfig`
- Added `psTimeAlloc`.
- Adding `errors` to `psVectorStats`.
- Adding `psRandom`, with three distributions (uniform, Gaussian, Poisson). This obsoletes `psGaussianDev`, and impacts `psLibInit` (no longer needs to seed the RNG).
- Changed `psImageRotate` to use radians instead of degrees.

C.6 Changes from Revision 08 (12 October 2004) to Revision 09 (15 November 2004)

- Updated `psSphereTransform`, according to bug 116.
- Removed mention of `libTAI` in the introduction.
- Fixed `psVectorHistogram` prototype to include `errors`.
- In `psVectorStats` and `psVectorHistogram`, the `errors` vector must be of the same type as the input values vector.
- Added `psLookupTable` section, and removed `psTimeTableInterpolate` (now redundant).
- `psTimeAdd` and `psTimeSubtract` merged to `psTimeMath`. Specified that time differences are expressed in SI seconds (i.e., not including leap seconds). Inputs to `psTimeMath` and `psTimeDelta` no longer need to be TAI — the functions will convert from and to UTC as required.
- modified the document name and PSDC number (and refs).
- cleaned up the FITS I/O functions: defined Alloc and Move functions, only use psFits for I/O.

- dropped deprecated psMetadataReadFits
- replaced `PS_META_ITEM_SET` with `PS_META_LIST`
- clarified use of x in psVectorFitSpline
- unified the form of evaluation functions for polynomials and splines
- clarified use of errors in stats function
- Changed return type of `psTraceReset`, `psLogSetFormat`, `psMetadataSetIterator`, `psMetadataItemPrint` to return type `bool`, so that errors won't be lost (bug 161).
- `psImageRebin` input types restricted; bug 198.
- Fixed up the LM minimization specification; bug 203.
- Removed errors from `psVectorFitSpline1D`.
- `psTraceReset` is to free memory used by `psTrace`.
- Added requirement on use of persistent memory — any use of persistent memory shall provide a function that frees the persistent memory.
- Added statement on use of `restrict`.
- In `psSpline1D`, renamed `domains` to `knots`.

C.7 Changes from Revision 09 (15 November 2004) to Revision 10 (30 November 2004)

- changed psFitsReadHeaderSet to return a psMetadata rather than just a psHash
- added METADATA ... END construction in psMetadataParseConfig
- added psArrayAdd function to Simple Arrays
- added utility functions `psMetadataLookupPtr`, `psMetadataLookupS32`, `psMetadataLookupF64`
- added `psHashToArray`
- re-organization of the astronomical images section: placed constructors with structures.
- added `XML functions`

C.8 Changes from Revision 10 (30 November 2004) to Revision 11 (21 January 2005)

- Changed names of `psSphereTransform` structure members to conform with ADD.
- Altered `psList` and `psListIterator` to match that in bug 249.
- added status element to psMetadataLookupTYPE utilities
- dropped psFitsCreateExt per discussion with rdd

- fixed error of psHash to psMetadata in psFitsReadHeaderSet
- added psFitsWriteImage
- changed psFitsWriteImageSection to psFitsUpdateImage
- added header entry to psFitsWriteTable
- added psFitsUpdateTable
- Updated `psSpline1D` to use a vector for the `knots`, and specified types.
- `psHistogram.nums` changed to type F32 to accomodate errors in the values.
- Synchronized use of `mask` throughout. A non-zero mask value means that the corresponding value shall not be used. Affected: polynomials (including plane transformations), minimization.
- Added `psPlaneAlloc, psSphereAlloc, psProjectionAlloc`.
- `psList`:
  - Adopted new names: `psListGetAndIncrement, psListGetAndDecrement` instead of `psListGetNext, psListGetPrev` — clearer description of functionality.
  - Changed `PS_LIST_HEAD = 0` and `PS_LIST_TAIL = -1`; negative indices specify an item relative to the end of the list.
  - The action of retrieving data from a list (with one of the three `psListGet` functions) is considered "borrowing" the reference, so no action is performed on the reference counter. Removed other mentions of reference counters, since these were not necessary.
  - `mutable` boolean in `psListIterator` specifies whether `psListAddAfter` or `psListAddBefore` may be used to modify the list through the iterator. This allows the use of `const psList` in `psListIteratorAlloc`.
- Added `psPlaneTransformInvert`, `psPlaneTransformCombine` and `psPlaneTransformFit`.
- Added `psAstrometryReadWCS`, `psAstrometryWriteWCS`, and `psAstrometrySimplify`.
- Added additional modes to `psImageInterpolateMode` to deal with variances: `PS_INTERPOLATE_BILINEAR_VARIANCE, PS_INTERPOLATE_BICUBIC_VARIANCE, PS_INTERPOLATE_SINC_VARIANCE`.
- Added `psImageTransform`.
- Added section of Database Functions

C.9 Changes from Revision 11 (21 January 2005) to Revision 12 (9 February 2005)

- In `psMatrixLUD`, changed `psVector *perm` to `psVector **perm` to allow the function to allocate the vector (bug 269).
- Removed in `psListAlloc`, "The number of iterators in the list is initially set to zero." (bug 271).
- Added `region` and `nSamples` to `psPlaneTransformCombine` and specified the algorithm.

- re-added psMetadata iterators, modified the APIs somewhat
- added the mode entry to psMetadataAddItem
- clarified the 'format' entry in psMetadataAdd
- added the function psMetadataAddV
- added the option flags `psMetadataFlags`
- changed psDBDumpCols() to return a psMetadata (bug 285)
- changed psDB to hold a MYSQL *
- changed psDBSelectColumnNum() to accept a psElemType parameter
- changed psDBSelectColumn() to return an psArray of strings
- renamed psDBUpdateRow() to psDBUpdateRows()
- renamed psDBDeleteRow() to psDBDeleteRows()
- renamed psDBInsertRow() to psDBInsertOneRow()
- add psDBInsertRows()
- Replaced `PS_INTERPOLATE_SINC` with `PS_INTERPOLATE_LANCZOS[234]`.
- noted that SLAlib wrapping will be replaced with our own functions
- dropped the Image I/O Functions section (functions moved to psFits)

C.10 Changes from Revision 12 (9 February 2005) to Revision 13 (30 March 2005)

- Added `color`, `magnitude` to `psAstrometryWriteWCS`, in order to convert a `psPlaneDistort` into a straight spatial polynomial.
- fix typo in psVector typedef
- add limit param to psDBSelectRows()
- change psDBUpdateRows() and psDBDeleteRows() to return signed values
- Made `psMemSetDeallocator` and `psMemGetDeallocator` public functions, and cleaned up description.
- Reworked `psLookupTable` functions so that the number of columns and their types are defined by a `scanf`-like format string. Added `psVectorsReadFromFile` and `psLookupTableImport` to generalise the reading of tabular data from a file.
- `psMetadataAddV` changed to use `va_list` parameter (bug 312).
- Modified `psImageTransform` in preparation for image combination.
- Added `psPixels` structure and related functions
- Added `psPlaneTransformDeriv`.

- Added `psImageGrowMask`.
- Added Earth Orientation Calculations Section
- Changes to the Time section:
  - Add `psTimeBulliten` enum
  - Add `leapsecond` member to `psTimeType`
  - Change `psTime.usec` → `psTime.nsec` (nanoseconds)
  - Minor reorganization and additional comments
  - Rename `psTimeGetTime()` → `psTimeGetNow()`
  - New rules for time system converstion
  - Rename `psTimeToLST()` → `psTimeToLMST()`
  - Rename `psTimeLeapSeconds()` → `psTimeLeapSecondDelta()`
  - Add `psTimeIsLeapSecond()`
  - ISO8601 format clarifications
  - Rename `psTimeToISOTime()` → `psTimeToISO()`
  - Add `psTimeFromUTC()`
  - Add `psTimeFromTT()`
  - Change `psTime` math rules
  - Change "Time Tables" to have IERS Bulliten A & B
  - Add Dates & Times Test Inputs appendix
- Adding logical operations (and, or) to `psBinaryOp`.
- Substantial reorganization:
  - Moved Metadata, Database, and XML sections to new section
  - Re-named Detector & Sky Coordinates to Linear & Spherical Coordinates
  - Moved Exposure and Observatory information out of 'Astronomical Images'
  - Moved Celestial Coordinate Systems out of 'Detector & Sky Coordinates'
  - Added Atmospheric Effects section, incorporating psGrommit and airmass functions from other sections)
  - Moved Fixed Pattern out of Astronomical Images

C.11 Changes from Revision 13 (30 March 2005) to Revision 14 (27 April 2005)

- Restrictions on the use of `malloc`, `calloc`, `realloc`, and `free` should not be unintentionaly imposed on 3rd party code.
- Add database support for "auto-incrementing"
- Changes to Configuration Files:
  - Add `UTC,UT1,TAI,TT` types
  - Change "multiple symbol" declaration format to `[keyword] MULTI`

  - Add Scoping Rules
  - Remove Configuration File Grammar appendix
  - Add Configuration File Test Inputs appendix
  - Rename `psMetadataParseConfig()` → `psMetadataConfigParse()`
  - Add `psMetadataConfigFormat()`
  - Add `psMetadataConfigWrite()`
- Add `PS_META_TIME` to `psMetadataType`
- Changed `psPixels` to vector-like array of `psPixelCoord` (bug 371).
- Added `psPixelsAlloc`, `psPixelsRealloc` and `psPixelsCopy` (bug 371).
- After conversation with GG:
  - Mention that Chebyshev domain should only be from -1 to +1, but won't restrict.
  - Type for `psSpline1D` is F32 only.
  - Matrix functions return `NULL` in the event of an error.
  - Changed API for `psVectorFitSpline1D`.
  - Removed pre-defined LM minimization functions; these will be defined in the Modules SDRS.
- defined `psEarthPole`, re-cast Earth Orientation Calculations to use it for inputs and outputs.
- minor name changes in Earth Orientation to match ADD changes.
- dropped TBD for `psFitsUpdateImage`
- `psAberration` return value defined.
- added `psArrayRemove` function (already exists in psArray.c)
- added `psVectorExtend` function
- changed inputs to `psImageSlice` to use `psRegion`
- changes involving `psRegion`
  - moved Image Regions section to beginning of Image Operations
  - dropped psImageSubsection (redundant now)
  - changed the following functions to use psRegion: psImageSubset, psImageTrim, psImageSlice, psImageCut
  - added clarification to meaning of psRegion x0,x1,y0,y1 values.
- removed psMetadata.ptype as per bug 313, dropped reference in psDB section.
- clarified the discussion of duplicate keys and the option flags in `psMetadata`
- Added `psLibFinalize` (bug 388).

C.12 Changes from Revision 14 (27 April 2005) to Revision 15 (15 June 2005)

- Bug 393:
  - add `PS_META_TIME` to `psMetadataType`
  - change `psTimeFromISO()` to accept a `psTimeType` to specifiy what time system the generated psTime object should be set to
- changes to "Database Functions"
  - change `psDBCreateTable()`'s definition to be clear about how to combine indexes and auto-increment
  - add `p_psDBRunQuery()`
- `psLibInit` does not seed the RNG (done in `psRandom`).
- Updated metadata iteration to use `psMetadataIterator`.
- Restoring `nbuckets` to `psHashAlloc`, since it appears to be required by the code.
- Adding `psArrayGet` and `psArraySet` since these are in the code.
- `psMatrixEigenvectors` returns type `psImage` (not a single vector).
- Specify that `col0,row0` shall be preserved with `psImageCopy`.
- Added functions `psErrorGetStackSize`, `psVectorRecycle`, `psVectorCopy`, `psMetadataItemAllocStr`, `psMetadataItemAllocF32`, `psMetadataItemAllocF64`, `psMetadataItemAllocS32`, `psMetadataItemAllocBool`, `psMetadataAddS32`, `psMetadataAddF32`, `psMetadataAddF64`, `psMetadataAddList`, `psMetadataAddStr`, `psMetadataAddVector`, `psMetadataAddImage`, `psMetadataAddHash`, `psMetadataAddLookupTable`, `psMetadataAddUnknown`, `psMetadataLookupF32`, `psBitSetClear`, `psRegionToString` (modified), `psImageRecycle`, `psImageFreeChildren` `psFitsGetExtName`, `psFitsSetExtName` already implemented.
- Renamed `psMaskToPixels` to `psPixelsFromMask` (bug 414).
- `psRegionFromString` returns `NaN` for any element that doesn't parse correctly (bug 416).
- unified the single and double precision polynomials
- re-ordered the psPolynomial*Alloc arguments
- changed the psPolynomials to be defined in terms of Norder not Nterms
- re-ordered the psVectorFitPoly* arguments
- added mask & maskValue ot psVectorFitPoly*
- changed the requirements on the input data vectors to psPolynomialFit and Eval
- added the psVectorClipFitPoly* functions
- SDRS API clean up and implementation re-synchronization
  - rename psFinalize() → psLibCleanup()

- change parameters expecting __LINE__ to type `unsigned int`
- change parameters expecting __LINE__ to be named `lineno`
- change `int` → `size_t` where appropriate
- change `psBool` → `bool`
- change `psU64` → `unsigned long long` where appropriate
- add `const` to parameters where appropriate
- remove `const` from parameters passed by value
- remove `const` from parameters that are actually being modified
- change `void *` → `psPtr` where appropriate
- change parameter names to be consistent with the current implementation
- change parameter names to be more consistent between related functions
- change parameter types to be more consistent with related data structures
- change `psF64` → `double` where appropriate
- change `psC64` → `complex double`
- rename functions/datatypes to abv. “allocate” as “alloc”
- rename functions/datatypes to abv. “memory” as “mem”
- rename functions/datatypes to abv. “function” as “func” (not “fcn”)
- no longer abv. “format” as “fmt”
- remove “my” from function parameters names
- add `psComparePtrFunc` function pointer
- change parameters expecting filenames to be named `filename`
- change parameters specifying the number of something to allocated to be named `nalloc`
- change parameters specifying the number of bytes to be allocated to be named `size`
- change parameters specifying the type of something to be named `type` where appropriate
- change `psImage` to use `psU32` rows & columns
- change parameter names to not use the words “width” or “height”
- rename `psHash.nbuckets` to `psHash.n`
- rename `psList.size` to `psList.n`
- rename `psFitsAlloc()` → `psFitsOpen()`
- add `psFitsClose()`
- change blurb about “Threads” to clarify requirements
- remove `lock` from `psList`
- change `psVector` to store it’s number of elements as an `long`
- change `psArray` to store it’s number of elements as an `long`
- change `psList` to store it’s number of elements as an `long`
- change `psListIterator` to store the number of elements on the list as an `long`
- change `psListIterator` to store it’s index as an `long`
- change `psHash` to store it’s number of elements as an `long`

  - change `psLookupTable` to store it's index as an `long`
  - change `psBitSet` to store it's size as an `long`
  - remove `psXMLDocFree()`
  - add `psXMLDocAlloc()`
- add `PS_META_NULL`
- add `NULL` keyword to "Configuration files"
- Added `psMemCheckTYPE` functions, for many values of `TYPE`.
- add `psMetadataItemAllocPtr()`
- add `psMetadataAddBool()`
- condense `psMetadataAddVector()`, `psMetadataAddImage()`, `psMetadataAddHash()`, `psMetadataAddLookupTable()`, `psMetadataAddUnknown()`, & `psMetadataAddMetadata()` into `psMetadataAddPtr()`
- add `psMetadataLookupStr()`
- add `psMetadataLookupBool()`
- add `psFitsType`
- add missing `psImageRecycle()` prototype (was already defined)
- add `psLookupStatusType`
- re-organized sections to place all data container before the operations.
- moved psRegion to subsubsection under psImage, out of image operations
- added psMinimizeGaussNewtonDelta
- defined gain ratio test in psMinimizeLMChi2
- added psRegionForImage
- added psImageSmooth
- added psVectorInit
- added psImageInit
- added fmt to psTrace
- added psTraceV
- unification of psTrace and psLogMsg syntax
- moved image hierarchy section to ModulesSDRS
- moved photometry section to ModulesSDRS
- Updating section on thread safety with new policy.

- add psFitsOpenFD()
- Removed example destructor function, since the current implementation does not follow it, but achieves the same result.
- Reorganised document: collections and mathematical structures are now separate sections; metadata, pixels lists and bit sets are collections; type information goes into system utilities.
- psMetadataType changed to psDataType (and so PS_META_* changed to PS_DATA_*), and expanded to include most psLib structures. PS_META_MULTI changed to PS_DATA_METADATA_MULTI.
- Added policy on psPtr, psString, Changed some void * to psPtr and some char * to psString, as appropriate. Noted that metadata functions must copy strings in to a psMetadataItem.
- Added psMask instead of unsigned int for mask values.
- Added psStringCopy, psStringNCopy, psStringAppend, psStringPrepend to match implementation.
- added psRegionForSquare
- added psImageMaskRegion, psImageKeepRegion
- added psImageMaskCircle, psImageKeepCircle
- Updated psFits functions following bug 412: added psFitsHeaderFromImage, psFitsHeaderFromTable, psFitsMoveEnd, psFitsDeleteExtName, psFitsDeleteExtNum, psFitsTruncate, psFitsHeaderValidate. Changed psFitsWriteImage and psFitsWriteTable to update the extname. Added mode options for psFitsOpen.
- add limit param to psDBDeleteRows()
- change p_psDBRunQuery() to accept a printf() style format
- added F (file:line) to psLogMsg and psTrace
- added psLogGetLevel

C.13 Changes from Revision 15 (15 June 2005) to Revision 16 (13 Sept 2005)

- Removed psLookupTableStatusType (see bugs 304, 454).
- Added psArrayElementsFree (already implemented).
- psMask changed to psMaskType (more clear).
- Use the region in psImageTransform to set the size of the output image (bug 453).
- list in psListIteratorAlloc is not const (bug 455).
- Clarified policy on signed/unsigned for memory allocation; psAlloc checks the allocation size against PS_MEM_LIMIT.
- psVector and psImage functions -Alloc, -Realloc and -Recycle changed to signed int.

- `nFail` in `psMetadataConfigParse` is `unsigned`.
- `timeval` changed to `struct timeval` throughout.
- Making various integers `unsigned` in the various `psPolynomial` types and `psSpline1D` (bug 460).
- Adjusted `psStats` to reflect robust / fitted statistics as defined in the ADD.
- Changed definition of `psDB` to abstract database type.
- clarified psTimeToLMST
- added cutCols, cutRows to psImageCut
- added remaining integer primatives to psMetadata
- added C++ compatibility specification
- Added S32, S64, U32, U64 to `psScalar` (bug 491).
- Added the `psCompare` functions.
- Removed F32 from polynomials — only use double precision.
- `psPixelCoord` now contains floats.
- `psImageSlice` uses `psPixels` to output coordinates.
- Added `psTimer` functions (prototypes by EAM).
- `psLibInit` and `psLibCleanup` changed to `psTimeInitialize` and `psTimeFinalize`. Calls to functions that require the time tables before calling `psTimeInitialize` shall produce an error.
- Added `psArgument` functions to provide simple argument handling.
- Changed return types of `psXMLDocToFile, psXMLDocToMem, psXMLDocToFD` to bool (bug 499).
- Clarified behaviour of `psLogSetDestination` and `psTraceSetDestination`.
- Split `psMetadataRemove` into `psMetadataRemoveKey` and `psMetadataRemoveIndex`.
- Added explanatory note about `psRegionFromString` and the FITS standard.
- add `psFitsOpenStream()`
- Removed `psFixedPattern`.
- Changed `format` option for `psMetadataAddS32,psMetadataAddF32,psMetadataAddF64,psMetadataAd`
- `psVectorFitSpline1D` now takes an input `psSpline1D`, and an optional error vector.
- Changed `complex double` to `complex` throughout.
- Added `psImageCountPixelMask`
- Added `psVectorCountPixelMask`
- Added `psVectorCreate`

- Added `psImageRow`
- Added `psImageColumn`
- moved `paramMask` to `psMinimization`
- added `paramMin` to `psMinimization`
- added `paramMax` to `psMinimization`
- added `paramDelta` to `psMinimization`
- adjusted `psMinimizeLMChi2` to drop `paramMask`

C.14 Changes from Revision 16 (13 Sept 2005) to Revision 17 (18 Oct 2005)

- moved param constraints to `psMinConstrain`
- changed type requirement on psMinimize functions to just F64
- changed psMinimize input arguments to accept weight not sigma
- changed psPolynomial masks from char to psU8.
- changed `psImage.col0,row0` changed from const.
- Removed `psMath`
- set `psPolynomial` order elements (nX, nY, etc)

C.15 Changes from Revision 17 (18 Oct 2005) to Revision 18 (06 Dec 2005)

- TBD-ed `psEOC_ParallaxFactor` — do not code yet.
- Removed `psFitsOpenFD` and `psFitsOpenStream` — not possible to implement these with `cfitsio`.
- `psFitsMoveEnd` changed to `psFitsMoveLast`, and moves to the last extension.
- `psFitsWriteImage` and `psFitsWriteTable` write to the end of the file.
- Added `psFitsInsertImage` and `psFitsInsertTable` to insert extensions; note that these are expensive.
- Changed `psFitsUpdateImage` to use `x0,y0` instead of a `psRegion`, which would make for an overconstrained system; specified origins.
- Removed `psFitsHeaderFromImage, psFitsHeaderFromTable`. The functionality will be handled by `psFitsWriteImage, psFitsWriteTable`.
- Added `psMetadataCopy`.
- Clarified `psPlaneTransformAlloc` and `psPlaneDistortAlloc`: arguments are polynomial order, not number of terms.
- Updated `psImageTransform` to use `psPixels` for `blankPixels` instead of a `psArray`.

- Added requirement on dynamic setting of mutex locks: `psMemThreadSafety` (see bug 586).
- Added return type for `psImageSmooth` (bug 588).
- clarified psRegion for subimages
- clarified units for `psProjectionAlloc`

C.16 Changes from Revistion 18 (06 Dec 2005) to Revision 19 (21 Feb 2006)

- minor `typedef` fixes
- sync `psImage`'s definition with the implementation in psLib
- convert `complex` declarations to be explicitly `double complex`
- change `psFitsReadTable()`'s `fits` param to be `const`
- changes to make image, subimage, and region consistent:
  - all region operations refer to the parent image coordinates
  - all functions which take a psImage and some coordinate refer to the parent coordinate frame
  - psRegionForImage : changed definition of region to refer to parent coords
  - psImageCountPixelMask : changed definition of region to refer to parent coords
  - psImageSubset : specified definition of region to refer to parent coords
  - psImageTrim : specified definition of region to refer to parent coords
  - psImageRow : specified definition of region to refer to parent coords
  - psImageColumn : specified definition of region to refer to parent coords
  - psImageSlice : specified definition of region to refer to parent coords
  - psImageCut : specified definition of region to refer to parent coords
  - psImageRadialCut : specified definition of region to refer to parent coords
- Added S8,S16,U8,U16,U32 to `psDataType` (bug 579).
- `psVector.n` and `psArray.n` to be initially set to zero
- `psArrayAlloc` and `psArrayRealloc` to initialise values
- add `*out` param to `psFitsReadHeaderSet()`
- sync `psElemType`, `psDataType`, & `psFitsType` with the implementation in pslib
- reorder `psPrecessMethod` to match the implementation in pslib
- set `psPolynomial?DAlloc()` functions params to have `type` first
- dropped unused 'stats' from `psLookupTableInterpolateAll`
- psImageSubset : specified that the input region and image need not overlap, but that the bound saturate to the limits of the input image
- allow `psMetadataItem` pointer types to have a value of `NULL`

C.17 Changes from Revistion 19 (21 Feb 2006) to Revision 20 (11 Apr 2006)

- Added `psFitsIsImage` and `psFitsIsTable`
- Added `psFitsReadImageCube`,`psFitsInsertImageCube`,`psFitsWriteImageCube`,`psFitsInsertIma` functions.
- Added `extname` to `psFitsWriteTable` and `psFitsInsertTable` to mirror the image versions.
- `psRegionFromString` adjusted to allow a `NULL` string region
- added `PS_META_NO_REPLACE`
- Added `psMetadataItemCopy`; altered `psMetadataCopy` slightly. Implementations exist.
- Added `psRegionAlloc` and `PS_DATA_REGION` for the rare occasions when we want to use a `psRegion` on one of the containers.
- Updated `psFitsValidateHeader`, `psFitsWriteImage`, `psFitsWriteTable`. Added `psFitsHeaderFromImage`, `psFitsHeaderFromTable`. See bug 733.
- Added `psLine` functions
- Added `psEllipse` functions
- Added `psStringSplit`
- Added `psStringStrip`
- Added `psMetadataPrint`
- Added `psMetadataItemTransfer`
- Added `psFitsWriteHeaderNotImage`
- Added `psImageFlip`
- Added `psImageJpeg` functions
- Added `psMetadataItemParse` functions
- Added `psImageBicube` functions
- Added `psRegionIsBad`
- Added `psRegionIsNaN`
- Added `psSparse` functions.

C.18 Changes from Revision 20 (11 Apr 2006) to Revision 21 (24 July 2006)

- updated description of `psStringCopy()` & `psStringNCopy`
- rename `psMemThreadSafety()` -¿ `psMemSetThreadSafety()`
- add `psMemGetThreadSafety()`
- add `psTimeToTM()` & `psTimeFromTM()`
- add `psTimeStrptime()`
- add `psTimeStrftime()`
- add `psDBLastInsertID()`
- add `psDBExplicitTrans()`
- add `psDBTransaction()`
- add `psDBCommit()`
- add `psDBRollback()`
- add `p_psDBRunQueryPrepared()`
- add `p_psDBFetchResult`
- add database subsubsecionts for “Transaction Control Database Functions” and “Low Level Database Functions”