# 1 Advisory I/O <*ppio.h*>

1. The header <***ppio.h***> declares two types and functions for performing block I/O in an efficient manner.

2. The type ***ppio_access_mode*** must be declared as an enumeration accepting at least the values ***PPIO_RDONLY***, ***PPIO_WRONLY***, and ***PPIO_RDWR***. Numerical equivalents are implementation-defined.

3. The ***ppio_iovec_t*** type is defined as a compound type, supporting at least the ***offset*** and ***length*** fields. The ***offset*** field represents a byte offset from the beginning of a mapping. The ***length*** field represents a byte length of raw data.

## 1.1 Treatment of error conditions

1. The behavior of all functions in <***ppio.h***> is specified for all values representable by its input arguments, except when stated otherwise. Errors are communicated via return values and/or the special thread-global variable ***errno***.

2. No function is required to modify the value of ***errno*** except as otherwise specified here. Client code must set the value of ***errno*** to 0 prior to library calls in order to properly distinguish between current and prior errors.

## 1.2 Mappings

1. Input and output to a variety of data sources is abstracted into logical data *mappings*, which are more generic in scope than traditional file access.

2. A mapping is an ordered sequence of bytes suitable for recording large data streams. Mappings are inherently binary. Under the same implementation, data read from a mapping shall be equivalent to data written to the mapping. Endianness conversions may or may not be required for data transfer between distinct implementations, but no other conversions shall be required to share data between implementations.

3. Allocation is inherently tied to a mapping. A mapping is always suitably aligned for any basic data type.

4. Mapping objects are created by *opening* a file, which may involve *creating* a file.

5. The address of the ***void*** object (mapping) may be significant; a copy of the object need not serve in place of the original.

6. Mapping objects should be assumed to be finite resources. Implementations must allow at least 128 mappings to be open at any one time.

7. A mapping open (***open_range***) must be paired with a corresponding close (***close_range***); it is an error for execution to terminate with outstanding mappings. For mappings created in write mode, corresponding file contents are undefined when a mapping lacks a close.

8. A *window* is a portion of a mapping which is visible to the client execution context. Mappings are not directly visible to application code. Windows should be considered extremely lightweight to create. An implementation must allow at least 16,384 windows to be available at any one time.

9. When a mapping is ***finished*** or destroyed, all associated mappings are automatically and implicitly invalidated. Client applications are not required to specifically terminate windows.

10. Mappings are modified through windows, but the results of such modifications may be delayed in time. Client code may not be sure that modifications to a mapping are visible until the mapping has been terminated via ***close_range***.

11. An implementation may allow creating multiple, concurrent mappings to a single, overlapping byte range. It is implementation-defined whether changes to the first mapping are visible in the second mapping. It is implementation-defined whether termination of one mapping invalidates windows from the second. Portable applications should expect closing a single mapping to invalidate all overlapping mappings.

## 1.3 Files

1. The *open_range* function
**Synopsis**

```
#include <ppio.h>
void* open_range(const char* filename, enum ppio_access_mode access,
                 off_t begin, off_t end);
```

**Description**

(a) A successful call to ***open_range*** creates and returns a new mapping which is associated with the given filename. Access is defined only for the byte offsets described by ***begin*** and ***end*** and the mode described by ***access***.

(b) Open a file with read mode (***PPIO_RDONLY*** for ***access***) fails if the file exists or cannot be read.

(c) The returned pointer may not be used in the LHS of a statement unless the file was opened in write mode (***PPIO_WRONLY*** or ***PPIO_RDWR*** for ***access***).

**Returns**

(a) The ***open_range*** function returns the beginning of the address range defined by ***begin*** and ***end***. If the operation fails, ***open_range*** returns a null pointer.

2. The ***readonev*** function
**Synopsis**

```
#include <ppio.h>
void* readonev(const void* map, const struct ppio_iovec_t *iv,
               size_t len);
```

**Description**

(a) The ***readonev*** function makes data available to the calling process. The ***map*** argument defines which mapping will be utilized. The ***iv*** argument is an array of ***len*** structures of type ***ppio_iovec_t***. The array describes a list of blocks the application will utilize in the near future. All ***offset***s in the array are relative to the byte range defined during creation of the mapping (i.e. the ***open_range*** call).

(b) It is an error to pass a 0-length array (***len=0***) to ***readonev***. Implementations must set termerrno to a nonzero hould this occur.

(c) The ***map*** argument must have been previously obtained via a call to ***open_range***. An implementation must detect and set ***errno*** to a nonzero value in this situation.

(d) If the ***map*** argument has previously been given as arguments to ***finished*** or ***close_range*** functions, the result is undefined. An implementation is encouraged to detect and report an error via ***errno***, when such an implementation can be performed efficiently.

(e) Every element of the ***iv*** array must be within the allowable bounds of the mapping, as defined during ***open_range***. An implementation may reject a ***readonev*** call if *any* element of the ***iv*** array violates this property (i.e. it may check all elements, not just the first). Implementations must set ***errno*** when rejecting an I/O operation.

**Returns**

(a) A successful call to ***readonev*** returns a *window*, a pointer to the beginning of the data specified via the first element of the ***iv*** array. Data access is defined only for the next ***iv[0].length*** bytes; the result of data access outside this range is undefined.

(b) An unsuccessful call to ***readonev*** always returns a null pointer.

3. The ***finished*** function
**Synopsis**

```
#include <ppio.h>
void finished(void* map);
```

**Description**

(a) The ***finished*** function indicates a process no longer needs a mapping. An implementation must return from the function before initiating I/O operations.

(b) Arguments to ***finished*** must have been acquired from a previous call to ***open_range***. It is implementation-defined whether an implementation reports this via ***errno***.

(c) Any windows created by ***readonev*** or ***readanyv*** are invalidated when the corresponding mapping is passed to ***finished***. Access to a window after this point is undefined; clients should expect abrupt program termination to be a likely occurrence.

**Returns**

(a) Nothing. ***finished*** cannot visibly fail. Error conditions result in no-operation.

4. The ***close_range*** function
**Synopsis**

```
#include <ppio.h>
int close_range(void* map);
```

**Description**

(a) The ***close_range*** function destroys a mapping. The ***map*** argument must have resulted from a previous call to ***open_range***.

(b) The function is heavyweight; any outstanding internal buffers must be flushed before the successful completion of ***close_range***.

(c) Conversely, it is undefined whether internal buffers are visible to other processes until after a call to ***close_range***.

**Returns**

(a) A successful call returns 0. An implementation may set ***errno*** to communicate error values, but must return a non-zero value regardless of the ***errno*** value.

## 1.4 Interaction with File I/O

(a) Applications may access files via both traditional I/O services as well as via mappings. Interactions between the two access modes which are not specifically defined in this section are undefined[1].

[1]Portable applications should restrict file access to one mode at a time. For example, use standard C routines to read or write metadata from/to a file, and then close the file and create a mapping from it for data access.