bpipe

Edition 1.0.17, for version 1.0.17
23 September 2013

Diab Jerius

Table of Contents

1 Copying

2 Introduction

The MST ray trace software suite is based upon a large number of individual programs which operate on a stream of photons. The programs are orchestrated into a production "pipeline" by passing the output of one program to the input of the next. This document describes the format of the data stream and a library of functions to access the data.

There are a few unique properties required of the data streams:

- Programs must be able to annotate the data.
- Programs must be able to add or delete data associated with the photons from the stream.
- Programs' knowledge of the contents of the data stream should be limited to only those data which they need.

The `bpipe` data format and library provide this functionality.[1] The data stream is organized into a single header packet and multiple data packets, each of which has the same format. The header packet both describes the format of the data packets and contains annotation data. Both the header and the data packets are extensible; the user can add or delete data from either.

The `bpipe` library provides tools that allow the program to manipulate the data stream. It provides for addition and deletion of data, for splitting the data stream, and for extracting appropriate data from the stream. It insulates applications from the exact layout of the data, and filters out unnecessary information about the contents.

2.1 Coding details

The package is `C`-callable. Due to its implementation its full capabilities are not accessible directly from Fortran, although glue routines are simple to construct. ANSI-C prototypes and all necessary `enums` and defines are available by including the file `bpipe.h`. It requires the `rbtree` and `linklist` packages.

Definition of field names (so that different programs know what's coming down the pike) must be enforced externally by some mechanism.

Those routines for which it is appropriate signal failure or an error condition by returning a special value and setting the global variable `bpipe_errno` to a value representing the error. The values are documented in the functions section (see Appendix A [Functions], page 21). Applications using the `bpipe` library **must** check for error returns!

[1] `bpipe` is a contraction of "binary pipe".

3 The Data Stream

Before accessing the data, it is important to understand how the data stream is organized and presented to the program.

The data stream consists of a single header packet, followed by multiple data packets. The header packet is split into three sections:

1. the header packet data definitions
2. the data packet data definitions
3. the header packet data

Data packets contain only data.

The data are organized into fields, and are identified by names. Data are present in the stream in binary form, and consist of scalars or tensors composed of intrinsic data types. Synthesized data types (structures or records) are not currently supported. The data types that are available (for both header and data packet data) are listed in Chapter 6 [Intrinsic Data Types], page 19.

3.1 Header packet data fields

Header data fields serve as annotation for the entire data stream, and are suitable for use as a history mechanism. To this end, more than one data field in a header packet may have the same name, permitting easy addition of history information. Fields with the same name are assigned an index number, starting at '`0`', and increasing in order of appearance.

3.2 Data packet data fields

Data fields in data packets are a bit more complicated to work with. Data packet field names *must* be unique. Data packets actually pass through three states as they are processed by the `bpipe` library:

input The image of the data packet as it exists in the input data stream.

core The image of the data packet in memory, in the form available to the program. This may differ from the input image because of additions, deletions, or resizes of data fields requested by the program

output The image of the data packet as written to the output stream. It may differ from the core image if the program has requested field deletions or resizing. Each output stream may have a different set of field deletions or resizes and thus different output images.

The program has great flexibility in determining how data packet fields are passed on. A few scenarios may provide illumination:

- The program simulates a detector, and accepts photons which have been projected to a given axial station. Since it no longer needs the direction cosines, it requests that they be deleted from the core image. Subsequent calculations require only percent polarization, so the full polarization information need not be output. It thus creates a

new data field for the summary information in the core image, and requests that the full polarization information be deleted from the output image.

- The program simulates a non-destructive beam mapper, recording the position and energy of photons at a particular axial station. Since it is non-destructive, it must pass on all of the information for each photon. The only information required for the diagnostic output is the photon's position and energy. It thus creates two output streams, and directs the interface routines to delete all but the position and energy from the output image for the diagnostic output stream, leaving the other stream as a carbon copy of the input.

The `bpipe` library takes care of the transformations from one state to the next. There are a few simple rules for the data packet field manipulations:

- additions are made to the core image only
- deletions are made to the input image and the core image
- deletions or resizes of the input image's fields are performed before the program has accessed the data packet
- deletions or resizes of the core image's fields are made when the core image is written to the output streams.

All manipulations of data packet field definitions take place before any data packets are passed to the application by the `bpipe` interface. Because all packets on a data stream have the same format, the definitions must be frozen before reading and writing data packets.

3.3 Data fields

Fields may have more than one dimension. A *matrix-spec* encapsulates the dimensionality and extents which fully qualify the size and shape of an *N*-dimensional matrix. Since creating a *matrix-spec* is a bit of a pain, the field addition and manipulation routines provide default behaviors which preclude most *matrix-spec* manipulation.

A *matrix-spec* is specified via the `C` structure `BPMatrix` which is defined in `bpipe.h`:

```
typedef struct
{
  size_t nd;            /* number of dimensions */
  size_t *extent;       /* nd length array with extents */
  size_t ne;            /* the total number of elements */
} BPMatrix;
```

The extent array is ordered such that lower dimensions' extents come first. This similar to Fortran's array ordering, and opposite to `C`'s. Note that the `ne` field is filled in by the `bpipe` library when verifying the *matrix-spec*; it should not be filled in by the application.

Fields may be resized, either by changing their extents or changing their dimensions. In addition to specifying the size of the new matrix, one must specify how the original matrix is mapped onto the new matrix. This is done by specifying the origin, dimensionality, and extents of the sub-matrix of the original data which is to be copied into the new matrix.

There are a number of utility routines which ease manipulation of `BPMatrix` structures: `bpipe_matrix_new`, `bpipe_matrix_new_va`, `bpipe_matrix_dup`, and `bpipe_matrix_delete`. Copies of fields' *matrix-specs* may be created with `bpipe_hdrf_matrix` or `bpipe_dpktf_matrix`, depending upon the type of the field.

Extent and offset arrays are simple arrays of type `size_t`, and can be created and deleted with `malloc` and free. However, a few convenience routines are provided: `bpipe_extent_new`, `bpipe_extent_new_va`, `bpipe_offset_new`, `bpipe_offset_new_va`.

4 Accessing the data stream

Accessing a data stream can be boiled down to the following steps

1. Create a `BPipe` object.
2. Attach an input stream to the `BPipe` object.
3. Attach one or more output streams to the `BPipe` object.
4. Add to or change the data in the header.
5. Select the fields in the data packets that you want transferred to the output streams. Create new data packet fields.
6. Map the core and output data packet images.
7. Process the data packets.
8. Close the `BPipe`.

4.1 Creating a `BPipe` object

The first step in accessing a `bpipe` data stream is to create a `BPipe` object with `bpipe_new`. A `BPipe` object is a structure which contains all of the information necessary to manage a data stream. `bpipe_new` returns a pointer to the newly created `BPipe` object, which is used by most of the utility routines. The application can create as many `BPipes` as is required. Multiple `BPipes` are necessary when reading from multiple data streams. If multiple output streams are being written with input taken from a single data stream, only one `BPipe` is required. Only if there are multiple output streams which have substantially different formats are multiple `BPipes` needed.

See Section A.1.5 [bpipe_new], page 24.

4.2 Attaching to input and output streams

If the application is acting as a sink or filter for data, it must attach a `BPipe` to an input data stream. If the program serves only as a source of data, it doesn't require an input stream. `bpipe_input` attaches a data stream to a `BPipe's` input channel and reads and parses the data stream's header packet. A `BPipe` cannot be attached to more than one input source.

If the program is a source or a filter of data, it must attach at least one output stream to the `BPipe`. The `bpipe_output` function creates an output channel and attaches a data stream to it. It returns a handle to the output channel that is used to identify the channel to other utility routines. More than one output stream may be connected to a single `BPipe`. If the input stream will be split into multiple output streams, it's far easier to attach multiple output streams than to create multiple `BPipes`, as it is possible to specify on a per output stream basis the data fields that are to be written.

At present input and output streams must be either files or the UNIX standard input or standard output streams. See Section A.1.1 [bpipe_input], page 21 and Section A.1.6 [bpipe_output], page 24.

4.3 Manipulating Header Fields

The header packet is automatically read and parsed when `bpipe_input` opens up the input data stream. A header field is accessed by its name and its index. An index of '`0`' will retrieve the first field with a given name; an index of '`BPHdrfIdx_LAST`' will retrieve the last field. The number of fields with a given name is returned by `bpipe_hdrf_n`. If the program needs to go through the list of header fields sequentially, repeated calls to `bpipe_hdrf_next` will step through the the header data fields in the order that they appear in the header. Fields added by the application before an input data stream is attached to the `BPipe` will appear first in the list; those added after the header is read will appear last.

See Section A.2.6 [bpipe_hdrf_n], page 34 and Section A.2.7 [bpipe_hdrf_next], page 35.

4.3.1 Field Information

The information which can be extracted from a header field are its matrix (`bpipe_hdrf_matrix`), its data type (`bpipe_hdrf_type`), and its data (`bpipe_hdrf_data`). Note that `bpipe_hdrf_data` returns a `C` pointer to the data, not the actual data itself. You are allowed to manipulate the data through the pointer, but you should not attempt to extend the data (for example, copying a larger array). This must be done with `hdrf_resize`. Changes made to the data after the header is written to the output streams will (obviously) not appear on the output streams.

4.3.2 Creation and Deletion

Header Fields are created with `bpipe_hdrf_add`, `bpipe_hdrf_array_add`, or `bpipe_hdrf_string_add`.

`bpipe_hdrf_add`
: This is the most general routine, and allows one to create multi-dimensional fields. It is also the easiest to use if a scalar field is desired.

`bpipe_hdrf_array_add`
: This routine will create a one dimensional array, without the need for explicitly creating a *matrix-spec*.

`bpipe_hdrf_string_add`
: This routine simplifies the creation of fields containing strings. Strings are always terminated with a '`\0`' character.

Fields are deleted with the `bpipe_hdrf_delete` routine. It can delete either a single field or all fields with a given name.

4.3.3 Resizing

Header fields are resized with the `bpipe_hdrf_resize` routine. Unlike data fields, which are resized upon input or output of a data packet, header fields are resized immediately. After a resize, any previous *matrix-spec* or data pointers (obtained with `bpipe_hdrf_data`) are invalid and must be retrieved again. (Note that since the `BPMatrix` structure returned by `bpipe_hdrf_matrix` is a *copy* of the field's *matrix-spec*, the application has the responsibility of freeing it.)

Resizing a field consists of specifying three pieces of information:

- The *matrix-spec* for the new matrix. If the dimensions of the new matrix are to be the same as the old matrix, it's easiest to get a copy of the old matrix via `bpipe_hdrf_matrix` and tweak the extents array:

```
BPMatrix *matrix = bpipe_hdrf_matrix( field-name, field-index );
if (NULL == matrix)
{
  (error handling code)
}
matrix->extent[0]++;
```

  This is also a good approach when changing a field's dimension:

```
BPMatrix *matrix = bpipe_hdrf_matrix( field-name, field-index );
if (NULL == matrix)
{
  (error handling code)
}
/* increment the number of dimensions and reallocate the extent array */
matrix->nd++;
matrix->extent =
   (size_t *) realloc( matrix->extent, matrix->nd * sizeof(size_t) );
if (NULL == matrix->extent)
{
  (error handling code)
}
matrix->extent[matrix->nd - 1] = 2;
```

  More general changes should be done by creating a new *matrix-spec* via `bpipe_matrix_new` or `bpipe_matrix_new`.

- The source sub-matrix to copy and its destination. The source sub-matrix is defined by its extents and its offset from the "origin" of the source matrix. Both are represented by one dimensional arrays of type `size_t`, and can easily be created by one of the utility routines (`bpipe_extent_new`, `bpipe_extent_new_va`, `bpipe_offset_new` and `bpipe_offset_new_va`). The destination offset is also a one dimensional `size_t` array.
- Initialization data. Before the data is copied from the old to new matrix, the elements in the new matrix are initialized. A default initialization is provided, which fills the matrix with appropriate values of '`0`'. The application can provide alternate data.

See Section A.2.8 [bpipe_hdrf_resize], page 36.

4.4 Manipulating Data Packet Fields

The manipulation of data packet fields is a bit trickier than that of header fields. As mentioned previously, data fields are mapped between input, core, and output states, and can be resized or deleted separately in more than one state. Because of this, information about a data packet field (such as where the actual data is located) is not available until all of the sizing, additions, and deletions have been finalized. The core and output images of a data packet are constructed by calling `bpipe_map` or `bpipe_map_alloc`. It is only after

this call that an application can request information about the position of particular fields within a data packet.

Additionally, memory for the data packet has to be allocated; usually this is done with the `bpipe_map_alloc` command. This memory will be freed when the `bpipe` is closed. If for some reason (double buffering of photons at the program level, for instance) the program is to handle allocation and deallocation, then `bpipe_map` should be used. It does not allocate space, but returns the size of a data packet. The application is then responsible for all allocation and deallocation.

4.4.1 Field Information

Unlike header fields, which always use a name and index combination to access field information, there are two methods of accessing data packet fields. If a single data packet is being read or written, and the same memory is used for all packets, *and* all that is required is a pointer to the data in the data packet, then the simplest approach is to use `bpipe_dpktf_datap`. Note that `bpipe_dpktf_datap` must be called *after* the core image is mapped (via `bpipe_map` or `bpipe_map_alloc`).

Here's an approach if you only need to look at one field in a data packet:

```
BPipe *bpipe;
DVector3 *pos;
void *core;

/* create bpipe, set input stream */
...

/* map data packet images */
if ( NULL == ( core = bpipe_map_alloc( bpipe, 1, NULL ) ) )
{
  (error handling code)
}

/* get pointer to position field in data packet */
if ( NULL ==
     ( pos = (DVector3 *) bpipe_dpktf_datap( bpipe, core, "position",
                                             BPDType_DVector3 ) ) )
{
  (error handling code)
}
```

This usually gets unwieldy if there is more than one data field that is being accessed. Here's a way around that:

```
DVector3 *pos;
DVector3 *dir;

pos = (DVector3 *) dpktf_data( bpipe, core, "position", BPDType_DVector3 );
dir = (DVector3 *) dpktf_data( bpipe, core, "direction", BPDType_DVector3 );
```

```
...

void *
dpktf_data( BPipe *bpipe, char *name, void *core_image, BPDataType *type )
{
  void *data = bpipe_dpktf_datap( bpipe, core_image, name, type );

  if ( NULL == data )
  {
    (error handling code)
  }

  return data;
}
```

Note that in these examples, space for a single data packet is allocated and used for each data packet, so the addresses of the data packet fields are constant. This is typical of most applications. In this situation all data offsets should be determined before reading in the data.

However, if multiple data packets are processed at once (e.g., several are read in at a time), it is necessary to redetermine the positions of the fields for each data packet processed. Since `bpipe_dpktf_datap` is a function, this can be expensive. The solution to this is to use a different access mechanism, via data packet field handles. Because one can obtaining a pointer to the field in the data packet directly from the handle (via a `C` preprocessor macro, rather than a function) this is a much more efficient way of accessing data from within a loop, especially if processing several data packets at once.

A data packet field handle is first retrieved via `bpipe_dpktf`. (This routine can also be used to determine if a data packet field already exists.) After mapping of the images, the offset of the data field in the core image is available, and can be retrieved with `bpipe_dpktf_offset`. Pointers into the core image of the data packet are available from `bpipe_dpktf_arr` and `bpipe_dpktf_data`. Field data is available only *after* a data packet has been read in via `bpipe_read_dpkts`. At that point the data can be retrieved with `bpipe_dpktf_val` or `bpipe_dpktf_valn`.

The following example illustrates a few ways of getting at the data in this situation.

```
BPipe *bpipe;
DpktField *dpktf;
DVector3 *pos;
void *core_image;
size_t core_image_size, n_pkts, pos_offset;

/* create bpipe, set input stream */
...

/* map data packet images */
```

```
if ( NULL == ( core_image = bpipe_map_alloc( bpipe, NPKTS, &core_image_size ) ) )
{
  (error handling code)
}

if ( NULL == ( dpktf = bpipe_dpktf( bpipe, "position" ) )
{
  (error handling code)
}

pos = (DVector3 *) bpipe_dpktf_data(dpktf, core_image);
pos_offset = bpipe_dpktf_offset( dpktf );

bpipe_errno = BPNOERROR;
while ( (n_pkts = bpipe_read_dpkts( bpipe, core_image, NPKTS)) > 0 )
{
   DVector3 *tpos = pos;        /* location of the "position" data */
   void *pkt = core_image;
   void *pkt_lim = core_image + n_pkts;

   /* tpos will point at the location of the "position" data in the
      current data packet.  this algorithm uses the fact that the
      data in a succeeding packet are always core_image_size bytes
      from those in the preceeding packet */
   for (; pkt < pkt_lim ;
          pkt += core_image_size, tpos += core_image_size )
   {
     double dist2;

     /* tpos may be calculated as above, or directly calculated
        here as follows */
     tpos = (DVector3 *) ((char *) pkt + pos_offset);

     dist2 = tpos->x * tpos->x + tpos->y * tpos->y + tpos->z * tpos->z;
   }
}

if ( bpipe_errno != BPNOERR )
{
  (error handling code)
}
```

Retrieval of all other field information requires a data packet field handle as well.

If the program needs to go through the list of fields sequentially, repeated calls to `bpipe_dpktf_next` will step through the the fields, retrieving handles for each field.

Some information is available before the mapping of data packet images by `bpipe_map` or `bpipe_map_alloc`. This includes a field's matrix (`bpipe_dpktf_matrix`), type (`bpipe_dpktf_type`). and name (`bpipe_dpktf_name`). The latter is useful when stepping through the list of fields via `bpipe_dpktf_next`. For an interesting application of this, see Section B.5 [bpipe_dump], page 122.

4.4.2 Creation and Deletion

Data packet fields are created with `bpipe_dpktf_add` and `bpipe_dpktf_array_add`. These are similar to the header field creation routines, except that the location of the data is not specified. This will be determined by `bpipe_map` or `bpipe_map_alloc`. New data fields are placed in the core image, and usually are written to the output streams.

Fields can be deleted from two places, the core image or an output image. Those which are deleted from the core image are never seen by the application. The application can delete fields from either all output images or from selected output images. Deletion is performed by the `bpipe_dpktf_delete` routine.

4.4.3 Resizing

Resizing of data packet fields is very similar to the resizing of header fields. There are, however, two routines available, one for resizing the core image of a field (Section A.3.13 [bpipe_dpktf_resize_core], page 50), and one for resizing an output image of a field (Section A.3.13 [bpipe_dpktf_resize_core], page 50). Data packet fields are resized when they are converted from input to core images or from core to output images. They may only be resized before the call to `bpipe_map` or `bpipe_map_alloc`. After a resize, any previously retrieved *matrix-spec* is invalid and must be retrieved again. (Note that since the `BPMatrix` structure returned by `bpipe_hdrf_matrix` is a *copy* of the field's *matrix-spec*, the application has the responsibility of freeing it.)

Apart from when the field is resized, the substantive difference between header fields and data packet fields is in the initialization of the new field matrices. Since the application has free access to the data packet core image before data is read in, it may initialize it without any help from the `bpipe` interface. To make life easier, the routine `bpipe_dpktf_init` will initialize a field without the application having to extract the *matrix-spec* for the field to determine the number of elements to initialize. For other routines which may be used for more complicated initializations, Section A.3.8 [bpipe_dpktf_init], page 46.

Initialization of the output images is done internally by the library, as the application has no access to that image. All data are initialized to appropriate values of '`0`'.

5 Internals

5.1 Data encoding

Data encoded in the `bpipe` format can only have one of the predefined datatypes (see Chapter 6 [Intrinsic Data Types], page 19). Because the data are transmitted in binary form, there are a number of incompatibilities which may arise when the data is read by a computer with a different CPU:

- floating point representation
- integer representation
- byte and/or word order
- datum alignment

There is at least one standard for data translation which can accomodate these differences (Sun Microsystem's XDR), but it isn't presently used by `bpipe`. Eventually `bpipe` will probably use it.

Currently, `bpipe` assumes that the data are in IEEE floating point format or two's complement integer format, and that they follow big endian byte and word order. Datum alignment is assumed to be on a byte boundary divisible by the size of the datum. Padding between elements of data structures is added on a per machine-basis.

5.1.1 Datatype setup

Data types are either atomic or conglomerate. Atomic types map to the standard C types of `int`, `double`, `unsigned int`. Integers are assumed to be have 32 bits. Conglomerate types are composed of atomic types or nested conglomerate types, and are mapped onto C structures. The set of available data types is fixed; it can be extended only by recompilation of the `bpipe` library. Data types are specified by user programs via a set of `enum` constants.

For each data type, `bpipe` keeps track of its size, and, if it's a conglomerate type, the sizes and offsets of its constituent types. When data is written to a `bpipe` stream, conglomerate types are resolved into their representative atomic types. Only the actual data is written to the stream – any padding which arises from the C structure representation of the data type is ignored. When data is read, any such padding is restored. Since the amount of padding is determined at compile time for each platform, the data stream is immune from the vagaries of internal structure alignment. The arrangement of elements of conglomerate types has been done to minimize the amount of internal padding in the C structure. This reduces the number of copy operations, as copies of adjacent memory locations are collapsed into a single copy operation.

The tables which define the sizes and offsets of the data types can be found in `datatypes_info.h`. Note that this file is automatically created by the program `mkdt`, which parses a restricted form of structure definitions (see `datatypes.h`).

6 Intrinsic Data Types

The intrinsic data types (specified by their `C enum` (`BPDataType`) constants and their `C` language equivalents are listed below. All of the equivalent structures are defined in the header file `datatypes.h`.

- `BPDType_double`

  ```
  double
  ```

- `BPDType_char`

  ```
  char
  ```

- `BPDType_int`

  ```
  int
  ```

- `BPDType_uint`

  ```
  unsigned int
  ```

- `BPDType_DVector2`

  ```
  typedef struct
  {
    double x;
    double y;
  } DVector2;
  ```

- `BPDType_DVector3`

  ```
  typedef struct
  {
    double x;
    double y;
    double z;
  } DVector3;
  ```

- `BPDType_IVector2`

  ```
  typedef struct
  {
    int x;
    int y;
  } IVector2;
  ```

- `BPDType_IVector3`

  ```
  typedef struct
  {
    int x;
    int y;
    int z;
  } IVector3;
  ```

- `BPDType_UIVector2`

  ```
  typedef struct
  {
  ```

```
  unsigned int x;
  unsigned int y;
}UIVector2;
```

- BPDType_UIVector3

```
typedef struct
{
  unsigned int x;
  unsigned int y;
  unsigned int z;
}UIVector3;
```

- BPDType_DComplex

```
typedef struct
{
  double r;
  double i;
}DComplex;
```

- BPDType_DCVector2

```
typedef struct
{
  DComplex q1;
  DComplex q2;
}DCVector2;
```

- BPDType_DCVector3

```
typedef struct
{
  DComplex q1;
  DComplex q2;
  DComplex q3;
}DCVector3;
```

Appendix A Functions

A.1 BPipe Manipulations

A.1.1 bpipe_input

attach an input stream to a binary pipe

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_input(
  BPipe *bpipe,
  const char *path
);
```

Parameters

`BPipe *bpipe`
: a pointer to the binary pipe structure

`const char *path`
: a pointer to a string describing the input file or device to associate with the input channel

Description

`bpipe_input` attaches an input stream to a binary pipe, opens it and reads the data stream's header.

The parameter `input` is a string describing the input file or device associated with the input channel. If it is the string '`stdin`', the input channel is associated with UNIX standard input.

Returns

It returns zero upon success, non-zero upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed path could not be opened. Check `errno` for more information.

`BPEBADPIPE`
: the header data definitions had errors

`BPEIOERROR`
: an error occurred whilst reading the pipe

`BPENOMEM` a memory allocation failed

A.1.2 bpipe_delete

Close the channels attached to a binary pipe and destroy the pipe.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_delete(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: the binary pipe to close

Description

This routine closes the input and output channels and deletes the associated data structures. It deletes the pipe's header parameter and data packet field binary trees. Finally, it frees the memory associated with the binary pipe structure itself.

A.1.3 bpipe_map

create transformation maps for data packet fields

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_map(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: a pointer to a binary pipe structure

Description

`bpipe_map` creates transformation maps between the input, core, and output images of a data packet. It should be called after all manipulations of data packet field definitions are complete. If the binary pipe has output channels, it must be called before writing the header information to the outputs, and should not be called afterwards. It must be called prior to any data packet input or output activity on the binary pipe, and should not be called after any such activity.

This routine should only be called once. It will not re-map an already mapped `Bpipe`.

Returns

It returns the size of the core image in bytes, which should be used by the caller to allocate space for the data packets. Upon error it returns '`0`' and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`

- the `BPipe` has already been mapped
- a data packet field's input image matrix was bogus
- the mapping between a field's input and core images or output and core images was bogus
- the `BPipe` has an input stream and data packets are defined on the input stream, yet all of the data packet fields have been deleted

`BPEBADPIPE`
: either all of the input data packet fields were deleted or all of an output channel's data packet fields were deleted.

`BPENOMEM` a memory allocation failed

A.1.4 bpipe_map_alloc

create transformation maps for data packet fields and allocate data packets

Synopsis

```
#include <bpipe/bpipe.h>

void *bpipe_map_alloc(
  BPipe *bpipe,
  int ndpkts,
  size_t *size_p
);
```

Parameters

`BPipe *bpipe`
: a pointer to a binary pipe structure

`int ndpkts`
: the number of data packets to allocate

`size_t *size_p`
: if not '`NULL`', a location where the size of the data packet will be stored

Description

`bpipe_map_alloc` is a convenience routine that calls `bpipe_map` to map the data packet, then allocates space for the requested number of data packets. The space allocated is managed by `BPipe`; the calling program should *not* free it. If the parameter `size_p` is not '`NULL`', it should point to an integer where the size of the data packet will be stored.

This routine should only be called once. It will not re-map an already mapped `BPipe`.

Returns

It returns a pointer to the allocated space upon sucess. Upon error it returns '`NULL`' and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`

- the number of data packets requested was non-positive
- the `BPipe` has already been mapped
- a data packet field's input image matrix was bogus
- the mapping between a field's input and core images or output and core images was bogus
- the `BPipe` has an input stream and data packets are defined on the input stream, yet all of the data packet fields have been deleted

`BPEBADPIPE`

either all of the input data packet fields were deleted or all of an output channel's data packet fields were deleted.

`BPENOMEM` a memory allocation failed

A.1.5 bpipe_new

Allocate and initialize a binary pipe structure.

Synopsis

```
#include <bpipe/bpipe.h>

BPipe *bpipe_new(void);
```

Description

`bpipe_new` allocates and initializes a binary pipe structure. It sets up the linked-lists and binary trees for the header parameter and data packet field definitions.

Returns

It returns a pointer to a dynamically allocated binary pipe structure. It prints a message to stderr and exits upon error. Upon failure it returns `NULL`, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM` a memory allocation failed

A.1.6 bpipe_output

create a binary pipe output channel

Synopsis

```
#include <bpipe/bpipe.h>

BPipeOutput *bpipe_output(
```

```
  BPipe *bpipe,
  const char *path
);
```

Parameters

`BPipe *bpipe`
: a pointer to the binary pipe structure

`const char *path`
: the path to the channel to attach to this output descriptor

Description

`bpipe_output` attaches an output stream and opens it.

The parameter `output` is a string describing the output file or device associated with the output channel. If it is the string '`stdout`', the output channel is associated with the standard output stream.

Returns

It returns an output handle upon success, `NULL` upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: The passed path could not be opened. Check `errno` for more information.

`BPEBADPIPE`
: The `BPipe`'s header has already been mapped or written. Output channels can only be opened before mapping or writing the header

`BPENOMEM` a memory allocation failed

A.1.7 bpipe_read_dpkts

Read data packets from the input channel of a binary pipe.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_read_dpkts(
  BPipe *bpipe,
  void *buf,
  size_t n_pkts
);
```

Parameters

`BPipe *bpipe`
: the binary pipe from which to read

`void *buf` pointer to a user-allocated memory buffer into which the data packet core images will be placed

`size_t n_pkts`
the number of data packets to read

Description

`bpipe_read_dpkt` reads a user specified number of data packets from a binary pipe input channel into a user-provided memory buffer. It converts the input data packet images into core data packet images.

Returns

It returns the number of data packets read. Upon error it returns '`0`' and sets `bpipe_errno` to `BPEIOERR`.

A.1.8 bpipe_write_dpkt_n

Write data packets to a binary pipe's output channels.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_write_dpkt_n(
  BPipe *bpipe,
  void *buf,
  size_t n_pkts,
  BPipeOutput *bpo
);
```

Parameters

`BPipe *bpipe`
the binary bpipe to which to write the packets

`void *buf` a pointer to a user-provided memory buffer containing the data packets to be written

`size_t n_pkts`
the number of data packets to write

`BPipeOutput *bpo`
the output channel to which to write the packets.

Description

`bpipe_write_dpkt_n` writes a user specified number of data packets to one or all of a binary pipe's output channels. It must be called after a call to `bpipe_write_hdr`. It converts the packets' core images into output images.

If `channel` is the constant `BPOutputChannel_ALL` (or `NULL`), the packets are written to all of the output channels. It is much more efficient to call `bpipe_write_dpkt` or `bpipe_write_dpkts` if the situation warrants it.

Returns

It returns zero upon success, non-zero upon failure.

A.1.9 bpipe_write_dpkt

Write a data packet to a binary pipe output channel.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_write_dpkt(
  BPipe *bpipe,
  void *buf,
  BPipeOutput *bpo
);
```

Parameters

`BPipe *bpipe`
: the binary bpipe to which to write the packets

`void *buf`
: a pointer to a user-provided memory buffer containing the data packets to be written

`BPipeOutput *bpo`
: the output channel to which to write the packets.

Description

`bpipe_write_dpkt` writes a single data packet to a single binary pipe output channel. Contrast it with `bpipe_write_dpkts` and `bpipe_write_dpkt_n`. It must be called after a call to `bpipe_write_hdr`. It converts the packets' core images into output images.

Returns

It returns zero upon success, non-zero upon failure. Upon failure `bpipe_errno` is set to `BPEIOERR`.

A.1.10 bpipe_write_dpkts

Write several data packets to a binary pipe output channel.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_write_dpkts(
```

```
  BPipe *bpipe,
  void *buf,
  size_t n_pkts,
  BPipeOutput *bpo
);
```

Parameters

`BPipe *bpipe`
: the binary bpipe to which to write the packets

`void *buf`
: a pointer to a user-provided memory buffer containing the data packets to be written

`size_t n_pkts`
: the number of data packets to write

`BPipeOutput *bpo`
: the output channel to which to write the packets.

Description

`bpipe_write_dpkts` writes one or more data packets to a single binary pipe output channel. Contrast it with `bpipe_write_dpkt` and `bpipe_write_dpkt_n`. It must be called after a call to `bpipe_write_hdr`. It converts the packets' core images into output images. It is more efficient to call `bpipe_write_dpkt` if there is only one data packet to be written.

Returns

It returns zero upon success, non-zero upon failure. Upon failure `bpipe_errno` is set to `BPEIOERR`.

A.1.11 bpipe_write_hdr

Write header information to a binary pipe's output channels.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_write_hdr(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: the binary pipe for which to output the header

Description

`bpipe_write_hdr` writes header information to a binary pipe's output channels. It must be called after a call to `bpipe_map` and before any data packets are written to the output channel(s). no manipulation of header parameter definitions and data or data packet field definitions should be done after invoking this function.

Returns

It returns zero upon success, non-zero upon failure. It sets `bpipe_errno` upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
- the header had already been written
- the `BPipe` hasn't been mapped (by `bpipe_map`)
- there are no output channels

`BPEIOERR` an I/O error occurred

`BPENOMEM` a memory allocation failed

A.2 Header Field Manipulations

A.2.1 bpipe_hdrf_add

Add a new data field to the header.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_hdrf_add(
  BPipe *bpipe,
  const char *name,
  BPDataType type,
  BPMatrix *matrix,
  void *data,
  int copy
);
```

Parameters

`BPipe *bpipe`
: binary pipe to which to add the field

`const char *name`
: the name of the field

`BPDataType type`
: the field's datatype

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

`BPMatrix *matrix`
: the matrix specification

`void *data`
: a pointer to the field's data

`int copy`
: if true, make a copy of the data

Description

This routine adds a data field to the header associated with the passed `BPipe`. Duplicate fields with the same name are allowed; they are given different index numbers. The new field is added to the end of the list of existing fields.

The passed name is duplicated. If the matrix specification is `NULL`, a matrix specification for a scalar is constructed. If the `copy` argument is true, a copy is made of the data pointed to by the `data` argument. Otherwise, the header field will refer directly to the memory it points to. This saves time and space for large data fields. if the passed data pointer is `NULL`, memory is allocated for it, regardless of the state of the `copy` flag.

If the calling procedure constructs a matrix specification (a `BPMatrix` structure), `bpipe_hdrf_add` uses this structure as is and directs the `BPipe` cleanup code to take responsibility for freeing it. The calling procedure should *not* free it, nor should it pass it to any other `BPipe` routine which takes responsibility for freeing it. It should not alter it in any way after passing it to `bpipe_hdrf_add`.

Returns

It returns zero upon success. Upon failure it returns non-zero, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the matrix specification was bogus
- the name was illegal
- the type specification was illegal

`BPEBADPIPE`
- the header was already been written
- the `BPipe` has been mapped

`BPENOMEM` a memory allocation failed

A.2.2 bpipe_hdrf_array_add

Add a one-dimensional array field to the header.

Synopsis

```
#include <bpipe/bpipe.h>
```

```
int bpipe_hdrf_array_add(
  BPipe *bpipe,
  const char *name,
  BPDataType type,
  void *data,
  size_t extent,
  int copy
);
```

Parameters

`BPipe *bpipe`
: binary pipe to which to add the field

`const char *name`
: the name of the field

`BPDataType type`
: the field's datatype

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

`void *data`
: a pointer to the field's data

`size_t extent`
: the extent of the array

`int copy` if true, make a copy of the data

Description

This routine adds a one dimesional array field to the header associated with the passed `BPipe`. Duplicate fields with the same name are allowed; they are given different index numbers. The new field is added to the end of the list of existing fields. The passed name specification is duplicated.

This routine simplifies the addition of one-dimensional arrays. Normally one must construct a matrix specification and call `bpipe_hdrf_add`. This routine takes care of that.

If the `copy` argument is true, a copy is made of the data pointed to by the `data` argument. Otherwise, the header field will refer directly to the memory it points to. This saves time and space for large data fields. if the passed data pointer is `NULL`, memory is allocated for it, regardless of the state of the `copy` flag.

Returns

It returns zero upon success. Upon failure it returns non-zero, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`

-
- the header was already been written
- the `BPipe` has been mapped

`BPENOMEM` a memory allocation failed

A.2.3 bpipe_hdrf_data

Get a pointer to a header field's data.

Synopsis

```
#include <bpipe/bpipe.h>

void *bpipe_hdrf_data(
  BPipe *bpipe,
  const char *name,
  size_t index
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

`size_t index`
: the field's index. set to `BPHdrfIdx_LAST` to select the last one.

Description

This routine looks up a header data field and returns a pointer to the data in the field. The calling routine may use this pointer to directly change the header field's data.

Returns

Returns a pointer to the data upon success. If the field doesn't exist, it returns `NULL`. If there's an error, it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the index specified was illegal

A.2.4 bpipe_hdrf_delete

Remove a header field.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_hdrf_delete(
  BPipe *bpipe,
  const char *name,
  size_t index
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

`size_t index`
: the field's index. set to `BPHdrfIdx_LAST` to delete the last one, `BPHdrfIdx_ALL` to delete all of them

Description

This routine will remove a field from the header and free the memory used by the header data. It does not renumber the indices of the fields which have the same name as the field. If requested, it can delete all of the fields of a given name.

Returns

It returns zero upon success, non-zero if the field didn't exist. If an error occurs (the index was bad), it returns non-zero and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the index specified was illegal

A.2.5 bpipe_hdrf_matrix

Retrieve a copy of a header data field's matrix specification.

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_hdrf_matrix(
  BPipe *bpipe,
```

```
  const char *name,
  size_t index
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

`size_t index`
: the field's index. set to `BPHdrfIdx_LAST` to select the last one.

Description

This routine makes a copy of the matrix specification associated with a header data field and returns a pointer to it.

Returns

This routine returns a pointer to a `BPMatrix` structure upon success. If the field doesn't exist, it returns `NULL`. If an error occurs, it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM` a memory allocation failed

`BPEBADARG`
: the index specified was illegal

A.2.6 bpipe_hdrf_n

Return the number of header fields with a given name.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_hdrf_n(
  BPipe *bpipe,
  const char *name
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

Description

Return the number of header fields with a given name.

Returns

It returns the number of header data fields which have the given name. If none match, returns '0'.

A.2.7 bpipe_hdrf_next

Get the name of the next header field in the header.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_hdrf_next(
  BPipe *bpipe,
  const char **p_name,
  size_t *p_index,
  void **p_last
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this data is associated

`const char **p_name`
: a place to store a pointer to the field name

`size_t *p_index`
: the index of the field

`void **p_last`
: the address of a pointer used to traverse the list

Description

It may be necessary to process the header data fields in the order that they appear in the header. This function provides the name and index of the next field. It requires that the user allocate three variables: a `char *` which will point to the name of the field, an `int` which will contain the field's index, and a `void *`, which is used by `bpipe_hdrf_next` to keep track of where it is. The last variable should be set to `NULL` the first time `bpipe_hdrf_next` is called. The invoking routine passes *addresses* of these three variables to this routine.

Do not add or delete header data fields in between calls to `bpipe_hdrf_next`.

Returns

It returns non-zero if a field is available, zero if the entire list has been traversed. The name and index are returned via the `p_name` and `p_index` arguments. Please note that the

pointer returned via `p_name` points to the original copy of the field's name. **Do not change this data!**.

A.2.8 bpipe_hdrf_resize

Resize the storage space associated with a header field.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_hdrf_resize(
  BPipe *bpipe,
  const char *name,
  size_t index,
  BPMatrix *matrix,
  size_t src_off[],
  size_t dst_off[],
  size_t extent[],
  void *init,
  size_t n_init
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

`size_t index`
: the field's index. set to `BPHdrfIdx_LAST` to select the last one.

`BPMatrix *matrix`
: the new matrix specification

`size_t src_off[]`
: offset of source submatrix to copy to destination. extent is equal to the dimensions of the source matrix.

`size_t dst_off[]`
: offset of destination submatrix. extent is equal to the dimensions of the destination matrix.

`size_t extent[]`
: the extents of the submatrix to copy. the extent of this array is equal to the dimensions of the source matrix.

`void *init`
: an array of data which will be used to initialize the new field array elements before copying data from the old field array. The data

must have the same type as the header field. Set to `NULL` to use the default initialization.

`size_t n_init`
: the number of initializing elements

Description

This routine resizes the storage space associated with a header data field. The user must have created a `BPMatrix` structure (using one of the `bpipe_matrix_new` routines or `bpipe_matrix_dup`) for the new representation of the data. This routine will allocate space for the new matrix, copy over the specified submatrix, and delete the old space.

The submatrix to be copied is specified via the `src_off`, `dst_off`, and `extent` arguments. They specify, respectively, the offset of the origin of the submatrix in the original matrix, its offset from the origin of the new matrix, and the extents of the submatrix. The submatrix offsets are arrays of integers, one element per dimension. Either may be `NULL`, indicating that the submatrix's origin coincides with that of the matrix. The extent argument is an array with as many elements as there are dimensions in the original matrix. It may be `NULL`, in which case the maximum extents possible will be automatically calculated.

Before any data is copied from the old field into the resized field, the new field is first initialized, so that any holes in the data will have deterministic values. By default the field is filled with zeroes of the appropriate type. The calling routine may also specify an array of data to be used to initialize the field. If the number of data elements in this array is less than that needed to initialize the field, it will be repeated.

Note that `bpipe_hdrf_resize` uses the matrix specification structure and all of the offset and extent arrays as is, directing the `BPipe` cleanup code to take responsibility for freeing them. The application should *not* free them, nor should it pass them to any other `BPipe` routine which takes responsibility for freeing them. After passing them to `bpipe_hdrf_resize`, it should not alter them in any fashion.

Returns

It returns zero upon success and non-zero if the data field doesn't exist. If there was an error it returns non-zero and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the new matrix specification was bogus
- the matrix copy wasn't consistent with the sizes of the old and new matrix specifications
- the passed index was out of bounds

`BPENOMEM` a memory allocation failed

A.2.9 bpipe_hdrf_string_add

Add a new string data field to the header.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_hdrf_string_add(
  BPipe *bpipe,
  const char *name,
  char *data,
  size_t len,
  int copy
);
```

Parameters

`BPipe *bpipe`
: binary pipe to which to add the field

`const char *name`
: the name of the field

`char *data`
: a pointer to the string

`size_t len`
: an optional length. Set to `0` to use the length of the passed string

`int copy`
: if true, make a copy of the data

Description

This routine adds a character array data field to the header associated with the passed `BPipe`. Duplicate fields with the same name are allowed; they are given different index numbers. The new field is added to the end of the list of existing fields. The passed name specification is duplicated.

This routine differs from `bpipe_hdrf_add` in that the passed data is expected to be a `C` style null-terminated string. It creates a one dimensional character array with an extent equal to the length of the passed string (including the terminating null). The calling procedure can specify an overriding string length, which must include space for the terminating null.

If the `copy` argument is true, a copy is made of the string pointed to by the `data` argument. In the case that this string is longer than the overriding length (if provided), it is truncated so that it will fit in the requested length and terminated with a '`\0`'. if the `copy` argument is false, the header field will refer directly to the memory it points to. This saves time and space for large data fields. However, if the requested length is shorter than the actual length of the string, it will write a '`\0`' in the passed string at the appropriate position.

If the passed data pointer is `NULL`, memory is allocated for it, regardless of the state of the `copy` flag.

Returns

It returns zero upon success. Upon failure it returns non-zero, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`

-
- the header was already been written
- the `BPipe` has been mapped
- `data` was `NULL` and the requested length was less than '`2`'.

`BPENOMEM` a memory allocation failed

Warnings

If the `copy` parameter is false, the `data` points to a string and the optional length is specified and is shorter than the actual length of the string, `bpipe_hdrf_string_add` will write a '`\0`' into the passed string. This may cause address violations if the string is a const array.

A.2.10 bpipe_hdrf_type

Return the datatype of a header field.

Synopsis

```
#include <bpipe/bpipe.h>

BPDataType bpipe_hdrf_type(
  BPipe *bpipe,
  const char *name,
  size_t index
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

`size_t index`
: the field's index. set to `BPHdrfIdx_LAST` to select the last one.

Description

Return the datatype of a header field.

Returns

It returns the data type of a header field, if it exists. If it doesn't exist, it returns `BPDType_NOTYPE`. Upon error it returns `BPDType` and sets `bpipe_errno`.

Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the index specified was illegal

A.3 Data Packet Field Manipulations

A.3.1 bpipe_dpkt_n

return the number of data packet fields

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_dpkt_n(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: binary pipe

Description

return the number of data packet fields

Returns

The number of data packet fields.

A.3.2 bpipe_dpktf_add

Add a data packet field to the core image of a data packet.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_add(
  BPipe *bpipe,
  const char *name,
```

```
  BPDataType type,
  BPMatrix *matrix
);
```

Parameters

`BPipe *bpipe`
: the binary pipe to which to add the new data packet field

`const char *name`
: the name of the new field

`BPDataType type`
: the data type of the new field

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

`BPMatrix *matrix`
: the field's matrix description

Description

This routine adds a new field to the definition of the core image of a data packet. If no similarly named field exists, a new field with the given name is created and placed at the end of the list of current fields. The name data is duplicated. If the matrix specification is `NULL`, a matrix specification for a scalar is constructed.

If the calling procedure constructs a matrix specification (a `BPMatrix` structure), `bpipe_dpktf_add` uses this structure as is and directs the `BPipe` cleanup code to take responsibility for freeing it. The calling procedure should *not* free it, nor should it pass it to any other `BPipe` routine which takes responsibility for freeing it. It should not alter it in any way after passing it to `bpipe_dpktf_add`.

Returns

It returns '`0`' upon success. If a field with the same name already exists, it returns '`1`'. It returns '`-1`' upon error, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the input matrix was bogus
- the passed name was empty or `NULL` or had illegal characters in it.

`BPENOMEM` a memory allocation failed

A.3.3 bpipe_dpktf_arr

Get a pointer to a data packet field's data.

Synopsis

```
#include <bpipe.h>

C-type *bpipe_dpktf_arr(
  DpktField *dpktf,
  void *core_image,
  C-type type
);
```

Parameters

`DpktField *dpktf`
: A data packet field handle obtained from either `bpipe_dpktf` or `bpipe_dpktf_next`.

`void *core_image`
: A pointer to memory which holds the core image of the data packet.

Description

This `C` preprocessor macro will be replaced by a pointer to the starting location in the specified core image of the data for the specified data packet field. The pointer will be cast to the `C` type specified by the `type` argument, which should be appropriate for the field's data type (see Chapter 6 [Intrinsic Data Types], page 19). The invoking routine may use this pointer to directly access the field's data.

This access method provides a cleaner way of retrieving structure members:

```
DVector3 *position = bpipe_dpktf_arr( dpktf, image, DVector3 );

position->x = ..
```

Each of the arguments to the macro is used only once, so complex expressions with side effects will not result in unsavory results. While this "function" has been coded as a macro for efficiency, if possible it should be used outside of a loop. All arguments are assumed to be legal.

A.3.4 bpipe_dpktf_array_add

Add a one-dimensional array field to the core image of a data packet.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_array_add(
  BPipe *bpipe,
  const char *name,
```

```
  BPDataType type,
  size_t extent
);
```

Parameters

`BPipe *bpipe`
: the binary pipe to which to add the new data packet field

`const char *name`
: the name of the new field

`BPDataType type`
: the data type of the new field

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

`size_t extent`
: the extent of the array

Description

This routine adds a one dimensional array field to the definition of the core image of a data packet. If no similarly named field exists, a new field with the given name is created and placed at the end of the list of current fields. The name data is duplicated.

This routine simplifies the addition of one-dimensional arrays. Normally one must construct a matrix specification and call `bpipe_dpktf_add`. This routine takes care of that.

Returns

It returns '`0`' upon success. If a field with the same name already exists, it returns '`1`'. It returns '`-1`' upon error, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the passed name was empty or `NULL` or had illegal characters in it.

`BPENOMEM` a memory allocation failed

A.3.5 bpipe_dpktf_data

Get a pointer to a data packet field's data.

Synopsis

```
#include <bpipe.h>
```

```
char *bpipe_dpktf_data(
  DpktField *dpktf,
  void *core_image,
);
```

Parameters

`DpktField *dpktf`
: A data packet field handle obtained from either `bpipe_dpktf` or `bpipe_dpktf_next`.

`void *core_image`
: A pointer to memory which holds the core image of the data packet.

Description

This `C` preprocessor macro will be replaced by a pointer to the starting location in the specified core image of the data for the specified data packet field. The invoking routine may use this pointer to directly access the field's data. In this case it should be first be cast to the appropriate `C` type (see Chapter 6 [Intrinsic Data Types], page 19). However, see Section A.3.3 [bpipe_dpktf_arr], page 42 for a cleaner approach. This routine is generally used when the pointer need not be cast to the appropriate type (for example, when using `bpipe_sprintf`).

Each of the arguments to the macro is used only once, so complex expressions with side effects will not result in unsavory results. While this "function" has been coded as a macro for efficiency, if possible it should be used outside of a loop. All arguments are assumed to be legal.

A.3.6 bpipe_dpktf_datap

return a pointer to a data packet field's data.

Synopsis

```
#include <bpipe/bpipe.h>

void *bpipe_dpktf_datap(
  BPipe *bpipe,
  void *core_image,
  const char *name,
  BPDataType type
);
```

Parameters

`BPipe *bpipe`
: the binary pipe that contains the data packet

`void *core_image`
: a pointer to the core image of the data packet

`const char *name`
: the name of the field

`BPDataType type`
: the suspected data type of the field

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

This routine returns a pointer to the location in the specified core image of the specified data packet field. It is passed the name of the field (rather than a handle to it, as required by `bpipe_dpktf_data`) and the required data type. If the passed type does not match the actual type in the data packet, an error is returned.

Unlike `bpipe_dpktf_data`, `bpipe_dpktf_datap` is a function, not a C preprocessor macro.

Returns

It returns a pointer to the data in the data packet. Upon error it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the requested data packet field doesn't exist

`BPEBADPIPE`
: the data types didn't match

A.3.7 bpipe_dpktf_delete

Delete a data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_delete(
  BPipe *bpipe,
  const char *name,
  BPDataSite site,
  BPipeOutput *channel
);
```

Parameters

`BPipe *bpipe`
: the binary pipe with which the data packet is associated

`const char *name`
: the name of the field to delete

`BPDataSite site`
: the data site from which to delete the field

  Possible values for a `BPDataSite` are as follows: `BPDSite_INPUT`, `BPDSite_CORE`, `BPDSite_OUTPUT`

`BPipeOutput *channel`
: the output channel for which to delete this field. (only relevant for deletion from output sites)

Description

This routine deletes a field from the definition of either the core or output images of a data packet. The `site` argument specifies the data site from which to delete the field, and should be either of the constants `BPDSite_CORE` or `BPDSite_OUTPUT` (denoting the core and output images, respectively). In the latter case, it is deleted from the specified output channel. If the output channel is the constant `BPOutputChannel_ALL` (or `NULL`), the field is deleted from all output channels.

Note that once a field is deleted from the core image, it's gone! If a field of the same name is added, it's a completely new field, not the one that was in the input image.

Returns

It returns '`0`' if the deletion was successful, '`1`' if the field doesn't exist. Upon error it returns '`-1`' and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the matrix specification was bogus
- the header was already been written
- the `BPipe` has been mapped

`BPENOMEM` a memory allocation failed

A.3.8 bpipe_dpktf_init

Initialize a data packet field's core image.

Synopsis

```
#include <bpipe/bpipe.h>
```

```
void bpipe_dpktf_init(
  DpktField *field,
  void *dpkt,
  void *init
);
```

Parameters

`DpktField *field`
: the field to initialize

`void *dpkt`
: the core image, allocated by the user after calling `bpipe_map`

`void *init`
: an (optional) initialization data element. Set to `NULL` to use the default initialization value

Description

This routine will replicate a datatype initialization structure throughout the core image of a data packet field. It is a simple utility routine which replaces calls to `bpipe_dpktf_matrix`, and `bpipe_datatype_init` or `bpipe_memfill`. If the user wants more control over how the data is initialized, these functions should be called instead.

This routine will fill a data packet's field image with either default initialization values (zeroes of the appropriate type) or the data from a user provided data element (which will be replicated for each element in the data packet field's array). It must be called *after* `bpipe_map` has been called.

A.3.9 bpipe_dpktf_matrix

retrieve a copy of one of a data packet field's matrix descriptions

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_dpktf_matrix(
  DpktField *dpktf,
  BPDataSite site,
  BPipeOutput *channel
);
```

Parameters

`DpktField *dpktf`
: a pointer to the data packet field structure from which to extract the matrix

`BPDataSite site`
: the image site the matrix spec refers to

Possible values for a `BPDataSite` are as follows: `BPDSite_INPUT`, `BPDSite_CORE`, `BPDSite_OUTPUT`

`BPipeOutput *channel`
: the output channel handle if requesting matrix spec for an output channel

Description

This routine makes a copy of the matrix specification associated with the specified image of a data packet field and returns a pointer to it.

Returns

This routine returns a pointer to a `BPMatrix` structure upon success. If an error occurs, it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- an input matrix was requested and none existed.
- the passed site was bogus

`BPENOMEM` a memory allocation failed

A.3.10 bpipe_dpktf_name

Retrieve the name of a data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

const char *bpipe_dpktf_name(DpktField *dpktf);
```

Parameters

`DpktField *dpktf`
: a pointer to the data packet field

Description

Retrieve the name of a data packet field.

Returns

This routine returns a pointer to the name of a data packet field, if it exists. If it doesn't exist, it returns `NULL`. Please note that the pointer returned points to the original copy of the field's name. **Do not change this data!**.

A.3.11 bpipe_dpktf_next

Return a handle to the next core data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

DpktField *bpipe_dpktf_next(
  BPipe *bpipe,
  void **p_last
);
```

Parameters

`BPipe *bpipe`
: the binary pipe with which the field is associated

`void **p_last`
: the address of a pointer used to traverse the list

Description

This routine is used when traversing the list of data packet fields. It requires an external pointer which it uses to keep track of where it is in the list. The first call whould be done with that pointer set equal to `NULL`. Note that the *address* of the pointer is passed to `bpipe_dpktf_next`, not the pointer itself. The value shouldn't be changed between calls to this routine. Additions or deletions should not be made to the list in between calls to this routine.

Returns

It returns a pointer to the data packet field's structure. This is used by the data packet field information extraction routines. If the entire list of data packet fields has been traversed, it returns `NULL`.

A.3.12 bpipe_dpktf_offset

Get the offset of a data field in a data packet.

Synopsis

```
#include <bpipe.h>

size_t bpipe_dpktf_offset(
  DpktField *dpktf,
);
```

Parameters

`DpktField *dpktf`
: A data packet field handle obtained from either `bpipe_dpktf` or `bpipe_dpktf_next`.

Description

This `C` preprocessor macro will be replaced by the offset into a data packet's core image of the specified data packet field.

Each of the arguments to the macro is used only once, so complex expressions with side effects will not result in unsavory results. All arguments are assumed to be legal.

A.3.13 bpipe_dpktf_resize_core

Resize the core image of a data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_resize_core(
  DpktField *dpktf,
  BPMatrix *matrix,
  size_t src_off[],
  size_t dst_off[],
  size_t extent[]
);
```

Parameters

`DpktField *dpktf`
: a pointer to the data packet field to resize

`BPMatrix *matrix`
: specification for the core site's storage description.

`size_t src_off[]`
: An array containing the offset coordinates of the source submatrix to copy to the new site. The number of elements is equal to the dimensions of the source matrix. If it's `NULL`, the origin is assumed.

`size_t dst_off[]`
: An array containing the offset coordinates of the destination submatrix in the new site. The number of elements is equal to the dimensions of the target state's matrix. If it's `NULL`, the origin is assumed.

`size_t extent[]`
: An array containing the extents of the submatrix to copy. The number of elements is equal to the dimensions of the source matrix. If it's `NULL`, the largest submatrix which will fit is copied.

Description

This routine allows one to specify the size of a data packet field's core image and the mapping from its input image. The input state is mapped onto the core state by copying

a submatrix of the input data. The default mapping is to simply duplicate the entirety of the input matrix, without changing its extents or dimensions.

The field's core image size is determined by the passed matrix specification. It may be created with `bpipe_matrix_new` or `bpipe_matrix_new_va`, or may be extracted from an existing data packet field. The matrix specification may also by `NULL`, indicating that the core image should have the same size as the input image.

The mapping is composed of specifications of the offset and extent of the source submatrix and the offset of the destination submatrix. These are passed as arrays. If an offset array is `NULL`, the offsets in each dimension are set to '`0`'. If the submatrix extent is `NULL`, the largest possible submatrix, as determined by the sizes of the input and core matrices and the submatrix offsets, is copied.

This routine may be called multiple times for the same data field; previous mappings are replaced. To return to the default input to core mapping, the matrix specification and the submatrix offsets and extent should be passed as `NULL`.

Note that `bpipe_dpktf_resize_core` uses the matrix specification structure and all of the offset and extent arrays directly, directing the `BPipe` cleanup code to take responsibility for freeing them. The application should *not* free them, nor should it pass them to any other `BPipe` routine which takes responsibility for freeing them. After passing them to `bpipe_dpktf_resize_core`, it should not alter them in any fashion.

Note that this routine does not do the actual mapping, it just stores the mapping specifications. `bpipe_map` does the mapping. Once a `BPipe` is mapped with `bpipe_map`, resizing data packet fields has no effect.

Returns

It returns zero upon success, non-zero upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the passed data packet field doesn't have a input image
- the passed matrix is bogus
- the input and core matrix sizes and the submatrix offsets and extents are inconsistent

`BPENOMEM` a memory allocation failed

A.3.14 bpipe_dpktf_resize_output

Resize the output image of a data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_resize_output(
```

```
  DpktField *dpktf,
  BPMatrix *matrix,
  size_t src_off[],
  size_t dst_off[],
  size_t extent[],
  BPipeOutput *bpo
);
```

Parameters

`DpktField *dpktf`
: a pointer to the data packet field to resize

`BPMatrix *matrix`
: specification for storage description at destination site.

`size_t src_off[]`
: An array containing the offset coordinates of the source submatrix to copy to the new site. The number of elements is equal to the dimensions of the source matrix. If it's `NULL`, the origin is assumed.

`size_t dst_off[]`
: An array containing the offset coordinates of the destination submatrix in the new site. The number of elements is equal to the dimensions of the target state's matrix. If it's `NULL`, the origin is assumed.

`size_t extent[]`
: An array containing the extents of the submatrix to copy. The number of elements is equal to the dimensions of the source matrix. If it's `NULL`, the largest submatrix which will fit is copied.

`BPipeOutput *bpo`
: the output channel

Description

This routine allows one to specify the size of a data packet field's output image and the mapping from its core image. The core state is mapped onto the output state by copying a submatrix of the core data. The default mapping is to simply duplicate the entirety of the core matrix, without changing its extents or dimensions.

The field's output image size is determined by the passed matrix specification, which is duplicated. It may be created with `bpipe_matrix_new` or `bpipe_matrix_new_va`, or may be extracted from an existing data packet field. The matrix specification may also by `NULL`, indicating that the output image should have the same size as the core image.

The mapping is composed of specifications of the offset and extent of the source submatrix and the offset of the destination submatrix. These are passed as arrays. If an offset array is `NULL`, the offsets in each dimension are set to '`0`'. If the submatrix extent is `NULL`, the largest possible submatrix, as determined by the sizes of the core and output matrices and the submatrix offsets, is copied.

This routine may be called multiple times for the same data field; previous mappings are replaced. To return to the default core to output mapping, the matrix specification and the submatrix offsets and extent should be passed as `NULL`.

This routine should be called after *all* calls to `bpipe_dpktf_resize_core` for this data packet field have been made. The sizes of the core and output images must be known in in order to validate the mapping between them.

Note that `bpipe_dpktf_resize_output` uses the matrix specification structure and all of the offset and extent arrays as is, directing the `BPipe` cleanup code to take responsibility for freeing them. The application should *not* free them, nor should it pass them to any other `BPipe` routine which takes responsibility for freeing them. After passing them to `bpipe_dpktf_resize_output`, it should not alter them in any fashion.

Note that this routine does not do the actual mapping, it just stores the mapping specifications. `bpipe_map` does the mapping. Once a `BPipe` is mapped with `bpipe_map`, resizing data packet fields has no effect.

Returns

It returns zero upon success, non-zero upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
- the passed matrix is bogus
- the input and core matrix sizes and the submatrix offsets and extents are inconsistent

`BPENOMEM` a memory allocation failed

A.3.15 bpipe_dpktf_type

return the datatype of a data packet field

Synopsis

```
#include <bpipe/bpipe.h>

BPDataType bpipe_dpktf_type(DpktField *dpktf);
```

Parameters

`DpktField *dpktf`
: a pointer to the data packet field

Description

return the datatype of a data packet field

Returns

Returns the data type of a data packet field, if it exists. If it doesn't exist, it returns `BPDType_NOTYPE`.

Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

A.3.16 bpipe_dpktf

Return an access handle for a data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

DpktField *bpipe_dpktf(
  BPipe *bpipe,
  const char *name
);
```

Parameters

`BPipe *bpipe`
: the binary pipe with which the field is associated

`const char *name`
: the name of the field

Description

This routine is used to obtain an access handle for a data packet field required by routines which extract data from the data packet field structure.

Returns

It returns a pointer to the data packet field's structure. It returns `NULL` if the field doesn't exist.

A.3.17 bpipe_dpktf_val

Retrieve the value of a data packet field's data.

Synopsis

```
#include <bpipe.h>

type bpipe_dpktf_val(
  DpktField *dpktf,
  void *core_image,
```

```
  C type type
);
```

Parameters

`DpktField *dpktf`
: A data packet field handle obtained from either `bpipe_dpktf` or `bpipe_dpktf_next`.

`void *core_image`
: A pointer to memory which holds the core image of the data packet.

`C-type` `type`
: The `C` type of the field (i.e., `double`, `IVector2`, etc.) as listed in Chapter 6 [Intrinsic Data Types], page 19.

Description

This `C` preprocessor macro will be replaced by either the value of the data field, if it is a scalar, or the value of the field's first element, if it is an array. `core_image` should point to the memory buffer containing the core image of the data packet from which the field data is to be extracted. The `type` argument is the actual `C` type of the data, which is used to correctly de-reference the pointer to the data.

Each of the arguments to the macro is used only once, so complex expressions with side effects will not result in unsavory results. All arguments are assumed to be legal.

Returns

It returns the value of the data field if it is a scalar, or the value of the field's first element, if it is an array.

A.3.18 bpipe_dpktf_valn

Retrieve an element in data packet field data array.

Synopsis

```
#include <bpipe.h>

type bpipe_dpktf_valn(
  DpktField *dpktf,
  void *core_image,
  C type type,
  size_t n
);
```

Parameters

`DpktField *dpktf`
: A data packet field handle obtained from either `bpipe_dpktf` or `bpipe_dpktf_next`.

`void *core_image`
: A pointer to memory which holds the core image of the data packet.

`C-type` `type`
: The `C` type of the field (i.e., `double`, `IVector2`, etc.) as listed in Chapter 6 [Intrinsic Data Types], page 19.

`n`
: An index into the array.

Description

This `C` preprocessor macro will be replaced by the value of the `n`'th element in a data packet field array. `core_image` should point to the memory buffer containing the core image of the data packet from which the field data is to be extracted. The `type` argument is the actual `C` type of the data, which is used to correctly de-reference the pointer to the data.

In the event that the field data is represented in `C` by a structure, structure members should be referenced via the "." notation:

```
x = bpipe_dpktf_valn(dpktf, data, DVector2, 2).x;
```

This is rather clumsy; For a cleaner method, see See Section A.3.3 [bpipe_dpktf_arr], page 42.

Each of the arguments to the macro is used only once, so complex expressions with side effects will not result in unsavory results. All arguments are assumed to be legal.

A.4 Utility Functions

A.4.1 bpipe_check_field_name

Check that a field name is legal.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_check_field_name(const char *name);
```

Parameters

`const char *name`
: the name to check

Description

Field names must start with a character from the alphabet or an underscore and can contain only alphanumeric and underline characters. Field names beginning with an underscore are reserved for use by `bpipe`.

Returns

It returns zero if the name is legal, non-zero if it isn't

A.4.2 bpipe_data_copy

copy data from one location to another

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_data_copy(
  void *dst,
  void *src,
  BPMatrix *matrix,
  BPDataType type
);
```

Parameters

`void *dst` the destination

`void *src` A pointer to the data

`BPMatrix *matrix`
: the data's matrix specification

`BPDataType type`
: the data type

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

`bpipe_data_copy` copies data of a given type and matrix specification.

Returns

nothing.

A.4.3 bpipe_data_dup

allocate memory for and duplicate a contiguous list of objects

Synopsis

```
#include <bpipe/bpipe.h>

void *bpipe_data_dup(
  void *data,
  size_t n,
  size_t size,
```

```
  int create_empty
);
```

Parameters

`void *data`
: the buffer containing the data

`size_t n`
: the number of elements of data

`size_t size`
: the size of an element

`int create_empty`
: if true, allocate and clear memory if the parameter data is NULL

Description

`bpipe_data_dup` allocates a block of memory and copies a sequential list of objects to that memory. If the pointer to the original data is `NULL`, and the `create_empty` flag is non zero, a block of memory large enough to contain the specified data is created and cleared. If the flag is zero, `NULL` is returned.

Returns

It returns a pointer to a block of memory duplicating the original data. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPENOMEM`
: a memory allocation failed

A.4.4 bpipe_datatype_init

Fill memory with a datatype initialization structure.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_datatype_init(
  void *dst,
  BPDataType type,
  size_t n
);
```

Parameters

`void *dst`
: the memory to init

`BPDataType type`
: the type of the datatype

Possible values for a BPDataType are as follows: BPDType_char, BPDType_double, BPDType_int, BPDType_uint, BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

`size_t n` the number of instances of the datatype to init

Description

This routine will replicate a default datatype initialization structure throughout a region of memory. The default initialization is the appropriate value of '0'. This routine is useful for intializaing a core image of a data packet field to a known state.

A.4.5 bpipe_datatype_name

Attach a name to a BPDataType.

Synopsis

```
#include <bpipe/bpipe.h>

const char *bpipe_datatype_name(BPDataType type);
```

Parameters

`BPDataType type`
the datatype

Possible values for a BPDataType are as follows: BPDType_char, BPDType_double, BPDType_int, BPDType_uint, BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

Description

Attach a name to a BPDataType.

Returns

This routine will return a pointer to a string which contains the name of the datatype corresponding to the passed BPDataType value. If the passed value is bogus, it returns NULL. Please note that the pointer returned points to the original copy of the field's name. **Do not change this data!**.

A.4.6 bpipe_datatype_resolve

Identify a datatype.

Synopsis

```
#include <bpipe/bpipe.h>
```

```
BPDataType bpipe_datatype_resolve(const char *datatype);
```

Parameters

`const char *datatype`
: a string with the name of a datatype

Description

This routine will return a `BPDataType` corresponding to the passed string, which should contain the name of a datatype.

Returns

It returns `BPDType_NOTYPE` upon failure, else the code for the datatype.

Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

A.4.7 bpipe_extent_new

Allocate an extent array.

Synopsis

```
#include <bpipe/bpipe.h>

void *bpipe_extent_new(
  size_t nd,
  size_t init
);
```

Parameters

`size_t nd` the number of dimensions

`size_t init`
: the integer value of the extents

Description

Extent arrays are required by various `bpipe` support routines. This routine creates one and sets all of its elements to the passed value.

Returns

It returns a pointer to an array of integers, filled in with the specified value. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the number of dimensions or the extent was less than 1

`BPENOMEM`
: a memory allocation failed

A.4.8 bpipe_extent_new_va

allocate an extent array

Synopsis

```
#include <bpipe/bpipe.h>

void *bpipe_extent_new_va(
  size_t nd,
  ...
);
```

Parameters

`size_t nd`
: the number of dimensions

`...`
: the integer values of the extents. there must be `nd` values, and should be of type `int`

Description

Extent arrays are required by various `bpipe` support routines. This routine makes it easy to allocate one and fill it in.

Returns

It returns a pointer to an array of integers, filled in with the specified values. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the number of dimensions or one of the extents was less than 1

`BPENOMEM`
: a memory allocation failed

A.4.9 bpipe_matrix_crunch

remove all singleton dimensions

Synopsis

```
#include <bpipe/bpipe.h>
```

```
BPMatrix *bpipe_matrix_crunch(BPMatrix *matrix);
```

Parameters

`BPMatrix *matrix`
: Not Documented.

Description

This routine returns a copy of the passed matrix with all dimensions of extent one removed. For example, '`[1][2][1]`' becomes '`[2]`'.

Returns

Upon successful completion a pointer to the new matrix is returned. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPENOMEM` a memory allocation failed

A.4.10 bpipe_matrix_delete

free a matrix specification structure and its accompanying data.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_matrix_delete(BPMatrix *matrix);
```

Parameters

`BPMatrix *matrix`
: the matrix specification to be freed

Description

If the passed pointer is not `NULL`, the memory it points to is freed. If the dimensional extent field is not `NULL`, that memory is freed as well.

A.4.11 bpipe_matrix_dup

duplicate a `BPipe` matrix structure

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_matrix_dup(BPMatrix *src);
```

Parameters

`BPMatrix *src`
: the matrix to duplicate

Description

This routine duplicates a binary pipe matrix and its accompanying data. It does not perform special processing for `NULL` matrix pointers.

Returns

On successful completion, a pointer to a new matrix structure is returned. upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM`
: a memory allocation failed

A.4.12 bpipe_matrix_min

create a binary pipe matrix with the minimum number of dimensions and extent from a pair of binary pipe matrix.

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_matrix_min(
  BPMatrix *a,
  BPMatrix *b
);
```

Parameters

`BPMatrix *a`
: Not Documented.

`BPMatrix *b`
: Not Documented.

Description

This routine creates a binary pipe matrix which has the minimum number of dimensions and extent from a pair of binary pipe matrix argument. It allocates memory for the matrix structure and its associated data.

Returns

Upon successful completion a pointer to the new matrix is returned. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM`
: a memory allocation failed

A.4.13 bpipe_matrix_max

create a binary pipe matrix with the maximum number of dimensions and extent from a pair of binary pipe matrix.

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_matrix_max(
  BPMatrix *a,
  BPMatrix *b
);
```

Parameters

`BPMatrix *a`
: Not Documented.

`BPMatrix *b`
: Not Documented.

Description

This routine creates a binary pipe matrix which has the maximum number of dimensions and extent from a pair of binary pipe matrix arguments. It allocates memory for the matrix structure and its associated data.

Returns

Upon successful completion a pointer to the new matrix is returned. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM`
: a memory allocation failed

A.4.14 bpipe_matrix_new

create a binary pipe matrix specification

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_matrix_new(
  size_t nd,
  size_t extent[]
);
```

Parameters

`size_t nd` the number of dimensions in the matrix

`size_t extent[]`
: if non `NULL`, points to an extents array to be duplicated

Description

This routine creates a binary pipe matrix specification of a given dimensionality. It allocates memory for the matrix structure and its associated data. If a premade extents array is available, the parameter `extent` should point to it, and the routine will use it as is, directing the `BPipe` cleanup code to take responsibility for freeing it. The user should *not* free it, nor should it pass it to any other `BPipe` routine which takes responsibility for freeing it. If `extent` is `NULL`, the extents array is initialized to all ones.

Returns

Upon successful completion a pointer to the new matrix is returned. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM` a memory allocation failed

A.4.15 bpipe_matrix_new_va

create a binary pipe matrix specification

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_matrix_new_va(
  size_t nd,
```

```
  ...
);
```

Parameters

`size_t nd` the number of dimensions in the matrix

`...` the extents of the dimensions. there must be as many extents as there are dimensions

Description

This routine creates a binary pipe matrix specification of a given dimensionality. It allocates memory for the matrix structure and its associated data. To ease filling in the associated extents array, the extents are passed as a variable argument list.

Returns

Upon successful completion a pointer to the new matrix is returned. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM` a memory allocation failed

A.4.16 bpipe_matrix_squeeze

remove all high order singleton dimensions

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_matrix_squeeze(BPMatrix *matrix);
```

Parameters

`BPMatrix *matrix`
: Not Documented.

Description

This routine returns a copy of the passed matrix with all high order dimensions of extent one removed. For example, '`[1][2][3][1]`' becomes '`[1][2][3]`'.

Returns

Upon successful completion a pointer to the new matrix is returned. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPENOMEM` a memory allocation failed

A.4.17 bpipe_memfill

Fill a region of memory with data.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_memfill(
  void *dstp,
  size_t n_dst,
  void *srcp,
  size_t n_src,
  size_t size
);
```

Parameters

`void *dstp`
: the memory to fill

`size_t n_dst`
: the number of data elements in the destination

`void *srcp`
: the data to replicate

`size_t n_src`
: the number of data elements in the src

`size_t size`
: the size of a data element

Description

This routine will replicate user supplied data throughout a region of memory. It uses `memcpy`, and attempts to invoke it as few times as possible.

A.4.18 bpipe_offset_new

Allocate an offset array.

Synopsis

```
#include <bpipe/bpipe.h>

size_t *bpipe_offset_new(
  size_t nd,
```

```
  size_t init
);
```

Parameters

`size_t nd` the number of dimensions

`size_t init`
: the value to initialize the offsets to

Description

Offset arrays are required by various `bpipe` support routines. This routine makes it easy to allocate one. It creates an offset array with all offsets set to a passed value.

Returns

It returns a pointer to an array of integers, filled in with the specified value. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the number of dimensions was less than 1

`BPENOMEM` a memory allocation failed

A.4.19 bpipe_offset_new_va

Allocate an offset array.

Synopsis

```
#include <bpipe/bpipe.h>

size_t *bpipe_offset_new_va(
  size_t nd,
  ...
);
```

Parameters

`size_t nd` the number of dimensions

`...`
: the integer values of the extents. there must be `nd` values, and should be of type `int`

Description

Offset arrays are required by various `bpipe` support routines. This routine makes it easy to allocate one and fill it in.

Returns

It returns a pointer to an array of integers, filled in with the specified values. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the number of dimensions was less than 1 or one of the offsets was negative

`BPENOMEM`
: a memory allocation failed

A.4.20 bpipe_proc_def

Process a field definition line.

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrix *bpipe_proc_def(
  char *buf,
  char **name,
  BPDataType *type
);
```

Parameters

`char *buf`
: the buffer which contains the definition

`char **name`
: the address of a pointer which will point to the name of the variable

`BPDataType *type`
: storage type of parameter

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

This routine takes a header or data packet field definition and determines the name of the field, its type, and its storage requirements. Definitions have the format

<data type>[<white space>]<name>[<matrix definition>]

Blank lines should *not* be passed to this routine, as they will be flagged as errors. All leading and trailing white space should be removed before calling `proc_def`.

Returns

It returns a pointer to a `BPMatrix` structure if a definition was available and parsed correctly, `NULL` upon error.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADPIPE`
: the definition was bad in some manner

`BPENOMEM`
: a memory allocation failed

A.4.21 bpipe_sprintf

Format a `BPipe` data field for output.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_sprintf(
  char *s,
  void *data,
  BPDataType type,
  char *formats[]
);
```

Parameters

`char *s`
: a buffer to hold the formatted output. it should be large enough to hold any output from any data type

`void *data`
: a pointer to the data to be formatted

`BPDataType type`
: the data's `BPipe` datatype code

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

`char *formats[]`
: an array of pointers to formats, with `BPDType_num` elements. These should be in the order of the `enum`'s (see `bpipe.h`) The first element (corresponding to `BPDType_NOTYPE`, should contain an error message. If this argument is `NULL`, the default formats are used

Description

This routine will format a data field for output. It fills a user-provided array with characters representing the data, in much the same way that `sprintf` does. It uses default output formats for the various data types, which may be changed by providing a new array of output formats.

Returns

It returns the same value as does `sprintf`.

A.4.22 bpipe_strerror

Get an error message string.

Synopsis

```
#include <bpipe/bpipe.h>

char *bpipe_strerror(BPerrno errnum);
```

Parameters

`BPerrno errnum`
: Possible values for a `BPerrno` are as follows: `BPNOERROR`, `BPEBADARG`, `BPENOMEM`, `BPEBADPIPE`, `BPEIOERR`, `BPENOFILE`

Description

`bpipe_strerror` returns a character string which describes the passed `BPipe` error code. The passed string is static, so don't try to overwrite it.

Returns

A pointer to a character string.

A.5 Internal Functions

A.5.1 bpipe_check_matrix_copy

Ensure that a matrix copy request is legal.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_check_matrix_copy(
  BPMatrix *src,
  BPMatrix *dst,
  BPMatrixMap *map
);
```

Parameters

`BPMatrix *src`
: description of the source matrix

`BPMatrix *dst`
: description of the destination matrix

`BPMatrixMap *map`
: description of source to destination mapping

Description

This routine checks that the dimensions of the submatrix are compatible with the destination matrix, that the dimensional extents are consistent with the source and destination matrices, and that the offsets are consistent. It can deal with non-expanded map entries.

Returns

It returns zero if the request is legal, non-zero otherwise.

A.5.2 bpipe_datatype_copy

call a copy function with offsets for a data type's atomic elements

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_datatype_copy(
  size_t dst,
  size_t src,
  size_t nobjs,
  BPDataType type,
  BPXMap map,
  void *udata,
  int (*copy)(void *,size_t,size_t,size_t)
);
```

Parameters

`size_t dst`
: starting destination offset

`size_t src`
: starting source offset

`size_t nobjs`
: number of data objects to copy

`BPDataType type`
: type of the data

Possible values for a BPDataType are as follows: BPDType_char, BPDType_double, BPDType_int, BPDType_uint, BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

`BPXMap map`
: what type of mapping

  Possible values for a BPXMap are as follows: BPXMap_Input_to_Core, BPXMap_Core_to_Output, BPXMap_Core_to_Core

`void *udata`
: copy routine specific data

`int (*copy)(void *,size_t,size_t,size_t)`
: a function to copy the data

Description

bpipe_datatype_copy issues copy commands to copy the elements in a data type without inter-element padding. A routine must be provided with an interface similar to memcpy which will do the actual copying of data. It assumes that all of the passed arguments are correct.

Returns

It returns zero upon success, non-zero if a copy operation returned failure.

A.5.3 bpipe_datatype_init_output

Fill memory with a datatype initialization structure.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_datatype_init_output(
  void *dst,
  BPDataType type,
  size_t n
);
```

Parameters

`void *dst` the memory to init

`BPDataType type`
: the type of the datatype

  Possible values for a BPDataType are as follows: BPDType_char, BPDType_double, BPDType_int, BPDType_uint,

BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

`size_t n` the number of instances of the datatype to init

Description

This routine will replicate a default datatype initialization structure for an output image of a field. The default initialization is the appropriate value of '0'. It is somewhat inefficient, as the compacted version of the initialization structures should be created at compile time.

A.5.4 bpipe_datatype_raw_size

return the size (in bytes) of the raw storage requirements of a binary pipe data type.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_datatype_raw_size(BPDataType type);
```

Parameters

`BPDataType type`
the data type

Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

return the size (in bytes) of the raw storage requirements of a binary pipe data type.

Returns

It returns the size in bytes of the raw storage space required for an instance of the data. It does not include any structure padding; just the total space directly used by the structure elements. If the requested type doesn't exist, it returns '0'.

A.5.5 bpipe_datatype_size

Determine the size (in bytes) of the internal storage requirements of a binary pipe data type.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_datatype_size(BPDataType type);
```

Parameters

`BPDataType type`
: the data type

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

Determine the size (in bytes) of the internal storage requirements of a binary pipe data type.

Returns

It returns the size in bytes of the internal storage space required for an instance of the data. This includes any structure padding. If the requested type doesn't exist, it returns '`0`'.

A.5.6 bpipe_datatype_write

Write a data type specification to an output channel.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_datatype_write(
  IOchannel *channel,
  BPDataType type
);
```

Parameters

`IOchannel *channel`
: the output channel to which to write the specification

`BPDataType type`
: the data type

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

This routine writes a data type specification to the indicated output channel which is parseable by `proc_defs`.

Returns

It returns zero upon success, non-zero upon failure.

A.5.7 bpipe_dpkt_cleanup

Cleanup `BPipe` data packet structures

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_dpkt_cleanup(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: binary pipe

Description

This routine is called by `bpipe_delete` to cleanup data packet structures in the `BPipe`.

A.5.8 bpipe_dpkt_setup

initialize dpkt data structure

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpkt_setup(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: binary pipe

Description

This routine is called by `bpipe_new` to initialize the data packet structures in a new `BPipe`.

Returns

It returns zero upon success, non-zero upon failure. It sets `bpipe_errno` upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM` a memory allocation failed

A.5.9 bpipe_dpktf_matrix_override_cmp

Compare a data packet field to a matrix override structure.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_matrix_override_cmp(
  const void *d,
  const void *m
);
```

Parameters

`const void *d`
: name

`const void *m`
: data packet field

Description

This routine compares a data packet field pointer to that in a matrix override structure.

Returns

It returns '`-1`', '`0`', '`1`' if the pointer is respectively less than, equal to, or greater than that in the matrix override structure.

A.5.10 bpipe_dpktf_newP

add a data packet field to a data packet.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_dpktf_newP(
  BPipe *bpipe,
  const char *name,
  BPDataType type,
  BPMatrix *matrix,
  size_t offset,
  BPDataSite site
);
```

Parameters

`BPipe *bpipe`
: the binary pipe to which to add the data packet field

`const char *name`
: the name of the new field

`BPDataType type`
: the new field's data type

Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

`BPMatrix *matrix`
: a description of the storage requirements of the field

`size_t offset`
: offset from start of input object

`BPDataSite site`
: the data site where the field is to be located

Possible values for a `BPDataSite` are as follows: `BPDSite_INPUT`, `BPDSite_CORE`, `BPDSite_OUTPUT`

Description

Adds a new field to either the input or core images of a data packet field. It does not allow duplicate fields to be created. It creates the necessary structures, replicates the user data (name, type) and places the parameter at the end of the list of fields as well as in the binary table of fields. If the matrix specification is `NULL`, a matrix specification for a scalar is constructed.

`bpipe_dpktf_newP` uses the passed matrix structure as is, and directs the `BPipe` cleanup code to take responsibility for freeing it. The calling procedure should *not* free it, nor should it pass it to any other `BPipe` routine which takes responsibility for freeing it.

This is a private routine, and should be called only be bpipe internal routines. The public routine, `bpipe_dpktf_add`, calls this with the site set to `DataSite_CORE`.

Returns

It returns '`0`' upon success. If a field with the same name already exists, it returns '`1`'. It returns '`-1`' upon error, and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: one of the arguments was bogus

`BPENOMEM` a memory allocation failed

A.5.11 bpipe_dpktf_size_cmp

Compare two data packet fields based upon the size of their data types.

Synopsis

```
#include <bpipe/bpipe.h>
```

```
int bpipe_dpktf_size_cmp(
  const void *dpktf1,
  const void *dpktf2
);
```

Parameters

`const void *dpktf1`
: pointer to first data packet field

`const void *dpktf2`
: pointer to second data packet field

Description

This routine compares two data packet fields based upon the size of their data types. In the case of ties, it uses the fields' index to break them (in ascending order). The size comparison is in reverse order. It assumes that the types are legitimate.

Returns

Returns '`-1`', '`0`', '`1`' if the second is less than, equal to, or greater than the first, respectively.

A.5.12 bpipe_hdr_cleanup

Cleanup header structure and deallocate header fields.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_hdr_cleanup(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: binary pipe whose header is to be initialized

Description

This routine is called to cleanup a `BPipe` header. It frees all memory associated with the header (fields, etc.).

A.5.13 bpipe_hdr_setup

Initialize hdr data structure.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_hdr_setup(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: binary pipe

Description

This routine is called by `bpipe_new` to initialize the header data structures of a new `BPipe`.

Returns

It returns zero upon success, non-zero upon failure. It sets `bpipe_errno` upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM` a memory allocation failed

A.5.14 bpipe_iochannel_close

close an I/O channel

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_iochannel_close(IOchannel *ioc);
```

Parameters

`IOchannel *ioc`
: the I/O channel to close

Description

Closes an I/O channel.

A.5.15 bpipe_iochannel_delete

delete an I/O channel structure

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_iochannel_delete(IOchannel *ioc);
```

Parameters

`IOchannel *ioc`
: the channel to deallocate

Description

This routine deallocates the memory associated with an I/O channel. if the passed pointer is `NULL`, it returns quietly.

A.5.16 bpipe_iochannel_fgetrn

Read a record from an I/O channel.

Synopsis

```
#include <bpipe/bpipe.h>

char *bpipe_iochannel_fgetrn(
  char **buf,
  size_t *buf_len,
  IOchannel *ioc
);
```

Parameters

`char **buf`
: a two element array of pointers which will be set to point to the the buffers used to store the input string. the buffers are dyamically re-allocated, hence the need for the address of the user-supplied pointer. buf[0] should be NULL upon the first call with it.

`size_t *buf_len`
: the length of the buffers

`IOchannel *ioc`
: the channel from which to read the data

Description

This routine reads data from an I/O channel until a newline character is read or the channel returns `EOF`. It truncates the trailing newline character. It dynamically resizes its buffers to encompass the data, and can thus read any length record (up to the memory limits on the machine).

The calling procedure must provide a pointer to the buffer space and an integer of type `size_t` which will record the length of the buffer. Upon first calling `bpipe_iochannel_fgetrn`, the buffer pointer should be set to `NULL`. Note that the *addresses* of these variables are passed to `bpipe_iochannel_fgetrn`.

Returns

It returns `NULL` upon end of file, else a pointer to the input buffer. Upon error it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPENOMEM` a memory allocation failed

A.5.17 bpipe_iochannel_new

create and initialize an IOchannel structure

Synopsis

```
#include <bpipe/bpipe.h>

IOchannel *bpipe_iochannel_new(const char *path);
```

Parameters

`const char *path`
: a path to the file or device to use as the I/O channel

Description

This routine allocates an `IOchannel` structure, making a copy of the passed path to the file or device to use as the I/O channel. It does not open the I/O channel.

Returns

It returns a pointer to a dynamically allocated `IOchannel` structure. It returns `NULL` if `path` is empty or `NULL`, or if it couldn't allocate memory for the structure.. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPENOMEM` a memory allocation failed

A.5.18 bpipe_iochannel_open

open an I/O channel

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_iochannel_open(
  IOchannel *ioc,
  const char *mode
);
```

Parameters

`IOchannel *ioc`
: a pointer to the `IOchannel` to open

`const char *mode`
: the mode with which to open the channel, see `fopen` for details

Description

This routine opens a previous allocated I/O channel.

It recognizes the paths '`stdin`' and '`stdout`' as special, and connects the channel to `stdin` or `stdout` respectively. In these cases, mode is ignored.

Returns

It returns zero upon succes, non-zero if it cannot open the path. `errno` should be checked upon error (not `bpipe_errno`), as it is set by `fopen`.

A.5.19 bpipe_iochannel_read

read data from an I/O channel.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_iochannel_read(
  void *buf,
  size_t size,
  size_t n,
  IOchannel *ioc
);
```

Parameters

`void *buf` the destination buffer to fill

`size_t size`
: the size of an object to be read

`size_t n` the number of objects to read

`IOchannel *ioc`
: the I/O channel to read from

Description

This routine reads data from an I/O channel. Because of the idiosyncracies of the various OS's implementation of `fread`, it isn't as efficient as it can be, nor does it return as much information as might be available.

Returns

It returns the number of objects read. Upon error it returns '`0`' and sets `bpipe_errno` to `BPEIOERR`.

A.5.20 bpipe_iochannel_write

Write data to an I/O channel.

Synopsis

```
#include <bpipe/bpipe.h>

size_t bpipe_iochannel_write(
  const void *buf,
  size_t size,
```

```
  size_t n,
  IOchannel *ioc
);
```

Parameters

`const void *buf`
: the source buffer to write

`size_t size`
: the size of an object to write

`size_t n`
: the number of objects to write

`IOchannel *ioc`
: the I/O channel to write to

Description

This routine writes data to an I/O channel. Because of the idiosyncracies of the various OS's implementation of `fread`, it isn't as efficient as it can be, nor does it return as much information as might be available.

Returns

It returns the number of objects written. Upon error it returns '`0`' and sets `bpipe_errno` to `BPEIOERR`.

A.5.21 bpipe_matrix_copy

Copy all or part of an N-dimensional matrix to a destination matrix.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_matrix_copy(
  BPDataType type,
  BPXMap xmap,
  BPMatrix *src,
  BPMatrix *dst,
  BPMatrixMap *map,
  void *udata,
  int (*copy)(void *udata,size_t dest,size_t src,size_t nbytes)
);
```

Parameters

`BPDataType type`
: type of a datum

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`,

BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

`BPXMap xmap`
: what type of mapping

  Possible values for a `BPXMap` are as follows: `BPXMap_Input_to_Core`, `BPXMap_Core_to_Output`, `BPXMap_Core_to_Core`

`BPMatrix *src`
: description of the source matrix

`BPMatrix *dst`
: description of the destination matrix

`BPMatrixMap *map`
: the mapping from source to destination

`void *udata`
: copy routine specific information

`int (*copy)(void *udata,size_t dest,size_t src,size_t nbytes)`
: this will be called multiple times to copy the chunks of data

Description

This routine analyzes the structure of the matrix (or submatrix) to be copied and the destination matrix to ensure that the largest possible chunks of memory are transferred. It passes information about contigous chunks to a user supplied function, which is responsible for the actual copying operation. The user supplied function is called by `bpipe_datatype_copy`.

Returns

It returns zero upon success, non-zero otherwise. Upon error `bpipe_errno` is set.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the matrix copy request was inconsistent with the sizes of the source and destination matrices or there were non-positive extents.

`BPEIOERR` one of the copy operations failed

`BPENOMEM` a memory allocation failed

A.5.22 bpipe_matrix_map_delete

bpipe_matrix_map_delete - free a matrix map structure and its accompanying data.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_matrix_map_delete(BPMatrixMap *map);
```

Parameters

`BPMatrixMap *map`
: the matrix map to delete

Description

if the passed pointer is not `NULL`, the memory it points to is freed along with the memory pointed to be the non `NULL` fields in the structure.

A.5.23 bpipe_matrix_map_dup

duplicate an existing matrix map structure

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrixMap *bpipe_matrix_map_dup(
  BPMatrix *src,
  BPMatrix *dst,
  BPMatrixMap *src_map,
  int create_empty
);
```

Parameters

`BPMatrix *src`
: source matrix spec

`BPMatrix *dst`
: destination matrix spec

`BPMatrixMap *src_map`
: the matrix map to be duplicated

`int create_empty`
: if non zero, allocate and clear memory for `NULL` map structure entries

Description

this routine creates a new matrix map structure and duplicates all available data in the passed matrix map. if the flag `create_empty` is non-zero, memory will be allocated and cleared for the `NULL` entries in the source matrix map.

Returns

it returns a pointer to the new matrix structure upon success. if the passed matrix map pointer is `NULL`, it returns `NULL`. upon error `bpipe_errno` is set and `NULL` is returned.

Errors

upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM`
: a memory allocation failed

A.5.24 bpipe_matrix_map_expand

expand all `NULL` matrix map entries to create default maps

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrixMap *bpipe_matrix_map_expand(
  BPMatrix *src,
  BPMatrix *dst,
  BPMatrixMap *map
);
```

Parameters

`BPMatrix *src`
: source matrix spec

`BPMatrix *dst`
: destination matrix spec

`BPMatrixMap *map`
: the matrix map to expand

Description

Matrix map structures may contain `NULL` entries indicating that default maps are to be performed. Before creating copy maps these must be expanded. This routine will, if necessary, fill in `NULL` matrix map entries. In the case that there are `NULL` entries in the structure, a new matrix map structure will be created and all available data duplicated. If offsets are not provided, they are set to the origin if the matrices. If the copied matrix extents are not provided, the extents of the largest submatrix which can be copied are used.

If the pointer to the source matrix map is `NULL`, it is treated as if all of the matrix map entries were `NULL`. in the case that a new map is created, the old is destroyed.

The resultant map is *not* checked to see if it is legal.

Returns

It returns a pointer to the original matrix if nothing needed to be expanded, else a newly created matrix map structure. The original map is destroyed if a new is created. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed source or destination pointers are bogus

`BPENOMEM`
: a memory allocation failed

A.5.25 bpipe_matrix_map_new

allocate a matrix map structure and its associated arrays.

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrixMap *bpipe_matrix_map_new(
  BPMatrix *src,
  size_t src_off[],
  BPMatrix *dst,
  size_t dst_off[],
  size_t extent[]
);
```

Parameters

`BPMatrix *src`
: source matrix spec

`size_t src_off[]`
: offset of source submatrix to copy to destination. extent is equal to the dimensions of the source matrix.

`BPMatrix *dst`
: destination matrix spec

`size_t dst_off[]`
: offset of destination submatrix. extent is equal to the dimensions of the destination matrix.

`size_t extent[]`
: the extents of the submatrix to copy. the extent of this array is equal to the dimensions of the source matrix.

Description

A matrix map structure is allocated and its associated arrays (source and destination offsets and copied submatrix extents) are constructed using the passed data (if available). It does not automatically create the new map's associated arrays if they are `NULL`.

Note that `bpipe_matrix_map_new` uses the offset and extent arrays as is, directing the `BPipe` cleanup code to take responsibility for freeing them. The application should *not* free them, nor should it pass them to any other `BPipe` routine which takes responsibility for freeing them. After passing them to `bpipe_matrix_map_new`, it should not alter them in any fashion.

Returns

It returns a pointer to a newly create matrix map structure upon success. Upon error `bpipe_errno` is set and `NULL` is returned.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADARG`
: the passed argument is bogus or the matrix specification was illegal

`BPENOMEM`
: a memory allocation failed

A.5.26 bpipe_matrix_override_delete

Delete a matrix override structure.

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_matrix_override_delete(BPMatrixOverride *over);
```

Parameters

`BPMatrixOverride *over`
: the matrix override structure to delete

Description

This routine frees the memory allocated with a matrix override structure and the associated matrix description structure. if the passed pointer is `NULL`, it returns quietly.

A.5.27 bpipe_matrix_override_new

Create a matrix override structure.

Synopsis

```
#include <bpipe/bpipe.h>

BPMatrixOverride *bpipe_matrix_override_new(
```

```
  DpktField *dpktf,
  BPMatrix *src,
  size_t src_off[],
  BPMatrix *dst,
  size_t dst_off[],
  size_t extent[]
);
```

Parameters

`DpktField *dpktf`
: data packet field to which matrix descriptor belongs

`BPMatrix *src`
: src matrix

`size_t src_off[]`
: offset of source submatrix to copy to destination. extent is equal to the dimensions of the source matrix.

`BPMatrix *dst`
: destination matrix spec

`size_t dst_off[]`
: offset of destination submatrix. extent is equal to the dimensions of the destination matrix.

`size_t extent[]`
: the extents of the submatrix to copy. the extent of this array is equal to the dimensions of the source matrix.

Description

This routine allocates memory for an output channel matrix override structure. It uses the passed matrix description and offset and extent arrays and directs the `BPipe` cleanup code to take responsibility for freeing them. The calling procedure should *not* free them, nor should it pass them to any other `BPipe` routine which takes responsibility for freeing them. It assumes that the matrix description is legit.

Returns

Upon success it returns a pointer to the newly allocated override structure, else it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM` a memory allocation failed

A.5.28 bpipe_matrix_verify

determine if a matrix specification is valid.

Synopsis

```
#include <bpipe/bpipe.h>

int bpipe_matrix_verify(BPMatrix *matrix);
```

Parameters

`BPMatrix *matrix`
: the matrix specification to validate.

Description

The number of dimensions and the extent of the dimensions of a matrix specification are checked to determine if they are positive. the number of elements in the matrix are recalculated. If the passed pointer is `NULL`, the routine simply returns.

Returns

If the specification is valid it returns zero, else non-zero.

A.5.29 bpipe_output_delete

delete an output descriptor structure

Synopsis

```
#include <bpipe/bpipe.h>

void bpipe_output_delete(BPipeOutput *bpo);
```

Parameters

`BPipeOutput *bpo`
: the output descriptor structure to delete

Description

This routine frees the memory associated with an output descriptor structure and its associated data. The I/O channel must already have been closed.

A.5.30 create_output_map

Create an output map for an output channel.

Synopsis

```
#include <bpipe/bpipe.h>

static int create_output_map(
  void *data,
  void *udata
);
```

Parameters

`void *data`
: Not Documented.

`void *udata`
: Not Documented.

Description

This routine creates a core image to output image map for a particular output channel. It is an action routine called with the output channel as the passed data.

Returns

It returns zero upon success, non-zero upon error. It sets `bpipe_errno` upon error.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADPIPE`
: there were no output maps

`BPENOMEM` a memory allocation failed

A.5.31 datatype_memcpy

`memcpy` type process function for `bpipe_datacopy_init_output`

Synopsis

```
#include <bpipe/bpipe.h>

static int datatype_memcpy(
  void *udata,
  size_t dest,
  size_t src,
  size_t nbytes
);
```

Parameters

`void *udata`
: Not Documented.

`size_t dest`
: Not Documented.

`size_t src`
: Not Documented.

`size_t nbytes`
: Not Documented.

Description

This routine is a process routine for `bpipe_datatype_copy`. It does an immediate copy of the data chunk. The `udata` passed to it must be a character array with two elements. The first should point to the beginning of the destination data, the second should point to the beginning of the source data.

A.5.32 dpktf_delete

Delete a `DpktField` structure.

Synopsis

```
#include <bpipe/bpipe.h>

static void dpktf_delete(DpktField *field);
```

Parameters

`DpktField *field`
: the field structure to delete

Description

Free the memory associated with a data packet field specification structure, including its associated data (name, type, input, core, output, input-to-core and core-to-output fields).

If any pointers are `NULL` (including the passed pointer to the data packet field specification), they are skipped.

A.5.33 dpktf_name_cmp

Compare a name to a data packet field's name.

Synopsis

```
#include <bpipe/bpipe.h>

static int dpktf_name_cmp(
  const void *name,
  const void *dpktf
);
```

Parameters

`const void *name`
: name

`const void *dpktf`
: data packet field

Description

This routine compares a name to a data packet field's name using string collating sequence. The first parameter is the name, the second a pointer to the data packet field structure.

Returns

It returns '`-1`', '`0`', '`1`' if the name is respectively less than, equal to, or greater than the field's name.

A.5.34 dpktf_new

Create and initialize a data packet field.

Synopsis

```
#include <bpipe/bpipe.h>

static DpktField *dpktf_new(
  const char *name,
  BPDataType type
);
```

Parameters

`const char *name`
: name of data packet field

`BPDataType type`
: storage class of field

  Possible values for a `BPDataType` are as follows: `BPDType_char`, `BPDType_double`, `BPDType_int`, `BPDType_uint`, `BPDType_DVector2`, `BPDType_DVector3`, `BPDType_IVector2`, `BPDType_IVector3`, `BPDType_UIVector2`, `BPDType_UIVector3`, `BPDType_DComplex`, `BPDType_DCVector2`, `BPDType_DCVector3`

Description

This routine creates and initializes a data packet field structure. It duplicates the passed field name, and type. They are all assumed to be valid.

Returns

It returns a pointer to a dynamically allocated and initialized `DpktField` structure. Upon error it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM` a memory allocation failed

A.5.35 dpktf_node_name_cmp

compare two data packet fields by their names

Synopsis

```
#include <bpipe/bpipe.h>

int dpktf_node_name_cmp(
  const void *dpktf1,
  const void *dpktf2
);
```

Parameters

`const void *dpktf1`
: first data packet field

`const void *dpktf2`
: second data packet field

Description

This routine compares two data packet fields by their names, using string collating sequence

Returns

It returns '`-1`', '`0`', '`1`' if the first field's name is respectifely less than, equal to, or greater than that of the second field.

A.5.36 dpktf_output_destroy

Remove a data packet field from an output channel's deletion list.

Synopsis

```
#include <bpipe/bpipe.h>

static int dpktf_output_destroy(
  void *data,
  void *udata
);
```

Parameters

`void *data`
: Not Documented.

`void *udata`
: Not Documented.

Description

This routine is an action routine called when traversing the output channel list to remove a data packet field from all output channels' deletion lists.

A.5.37 fill_core_to_output_map

Given a data packet field, determine the copy operations necessary to map from core to output state.

Synopsis

```
#include <bpipe/bpipe.h>

static int fill_core_to_output_map(
  void *data,
  void *udata
);
```

Parameters

`void *data`
: Not Documented.

`void *udata`
: Not Documented.

Description

This routine is an action routine passed to the linked list package. It takes a passed data packet field and generates a list of copy operations which map the data packet field from its core state to its output state. It uses and updates a passed offset from the start of the output image.

Returns

It returns zero upon success, non-zero upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the data packet's input matrix was bogus

`BPENOMEM`
: a memory allocation failed

A.5.38 fill_dpktf_ll

add the next link in a list

Synopsis

```
#include <bpipe/bpipe.h>

static int fill_dpktf_ll(
  void *data,
  void *udata
);
```

Parameters

`void *data`
: Not Documented.

`void *udata`
: Not Documented.

Description

This routine is an action function called when parsing the list of data packet fields. It is passed a data field pointer and a list pointer (as user data) and inserts the field pointer into the list.

Returns

It returns zero upon success, non-zero if it couldn't insert the the data into the list.

A.5.39 fill_input_to_core_map

Given a data packet field, determine the copy operations necessary to map from input to core state.

Synopsis

```
#include <bpipe/bpipe.h>

static int fill_input_to_core_map(
  void *data,
  void *udata
);
```

Parameters

`void *data`
: Not Documented.

`void *udata`
: Not Documented.

Description

This routine is an action routine passed to the linked list package. It takes a passed data packet field and generates a list of copy operations which map the data packet field from its input state to its core state. It uses and updates a passed offset from the start of the core image.

Returns

It returns zero upon success, non-zero upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the data packet's input matrix was bogus

`BPENOMEM`
: a memory allocation failed

A.5.40 hdrf_all_delete

`SLinkList` node delete routine for header data fields.

Synopsis

```
#include <bpipe/bpipe.h>

static void hdrf_all_delete(
  void *field,
  void *ll
);
```

Parameters

`void *field`
: the field to delete

`void *ll`
: the parent `bpipe->hdrf_ll` list

Description

This routine will delete a header field structure and the associated link in the parent `bpipe` structure's `hdrf_ll` list.

A.5.41 hdrf_channel_read

Read data from an input channel to memory.

Synopsis

```
#include <bpipe/bpipe.h>

static int hdrf_channel_read(
  void *udata,
  size_t src,
  size_t dst,
  size_t nbytes
);
```

Parameters

`void *udata`
: Not Documented.

`size_t src`
: Not Documented.

`size_t dst`
: Not Documented.

`size_t nbytes`
: Not Documented.

Description

This is a callback routine for `bpipe_datatype_copy` which reads the requested number of bytes from the input channel in the file global variable `hdrf_channel_read_ioc` and deposits them in the memory pointed to by the `udata` parameter (with the appropriate offset as given by `bpipe_datatype_copy`. It ignores the `src` argument.

Returns

It returns zero upon success, non-zero upon failure

A.5.42 hdrf_channel_write

Write data from memory to an output channel.

Synopsis

```
#include <bpipe/bpipe.h>

static int hdrf_channel_write(
  void *udata,
  size_t src,
  size_t dst,
  size_t nbytes
);
```

Parameters

`void *udata`
: Not Documented.

`size_t src`
: Not Documented.

`size_t dst`
: Not Documented.

`size_t nbytes`
: Not Documented.

Description

This is a callback routine for `bpipe_datatype_copy` which writes the requested number of bytes from the memory pointed to by the passed `udata` parameter (with the appropriate offset as given by `bpipe_datatype_copy` to the output channel in the file global variable `hdrf_channel_output_ioc`. It ignores the `dst` argument.

Returns

It returns zero upon success, non-zero upon failure

A.5.43 hdrf_delete

Delete a header data field structure and its associated data.

Synopsis

```
#include <bpipe/bpipe.h>

static void hdrf_delete(HdrField *field);
```

Parameters

`HdrField *field`
: the header field structure to delete

Description

This routine deletes the memory associated with a header data field structure. It does *not* remove it from the header lists; this must be done prior to calling `hdrf_delete`. It is safe to call this routine with a `NULL` pointer.

A.5.44 hdrf_get

Get a pointer to a header field.

Synopsis

```
#include <bpipe/bpipe.h>

static HdrField *hdrf_get(
  BPipe *bpipe,
  const char *name,
  size_t index
);
```

Parameters

`BPipe *bpipe`
: binary pipe with which this field is associated

`const char *name`
: the field's name

`size_t index`
: the field's index. set to `BPHdrfIdx_LAST` to select the last one.

Description

Get a pointer to a header field.

Returns

It returns a pointer to the header field if it exists, `NULL` if it doesn't. Upon error it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the index specified was illegal

A.5.45 hdrf_index_cmp

Compare an index to that in a header field.

Synopsis

```
#include <bpipe/bpipe.h>

static int hdrf_index_cmp(
  const void *indexp,
  const void *field
);
```

Parameters

`const void *indexp`
: a pointer to the name

`const void *field`
: a pointer to the field

Description

Compare an index to that in a header field.

Returns

It returns '`-1`', '`0`', or '`1`' if the index is less than, equal to, or greater than, the header field's index, respectively

A.5.46 hdrf_ll_delete

`SLinkList` node delete routine for header data fields.

Synopsis

```
#include <bpipe/bpipe.h>

static void hdrf_ll_delete(void *node);
```

Parameters

`void *node`
: the first field to delete

Description

This routine will delete a header field structure and all of the sibling fields with the same name. It does *not* delete the link in the parent `bpipe` structure's `hdrf_ll` list.

A.5.47 hdrf_name_cmp

Compare a name to a header parameter's name using string collating sequence.

Synopsis

```
#include <bpipe/bpipe.h>

static int hdrf_name_cmp(
  const void *name,
  const void *field
);
```

Parameters

`const void *name`
: a pointer to the name

`const void *field`
: a pointer to the field

Description

Compare a name to a header parameter's name using string collating sequence.

Returns

It returns '`-1`', '`0`', or '`1`' if the name is less than, equal to, or greater than, the header parameters name, respectively

A.5.48 hdrf_new

Create and initialize a header data field.

Synopsis

```
#include <bpipe/bpipe.h>

static HdrField *hdrf_new(
  const char *name,
  BPDataType type,
  BPMatrix *matrix,
  void *data,
  int copy
);
```

Parameters

`const char *name`
: the field's name

`BPDataType type`
: the field's data type

Possible values for a BPDataType are as follows: BPDType_char, BPDType_double, BPDType_int, BPDType_uint, BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

`BPMatrix *matrix`
: the field's storage spec

`void *data`
: a pointer to the data

`int copy`
: if true, make a copy of the data, else just stash the pointer

Description

This routine creates and initializes a header data field structure. It duplicates the passed field name and type. It assumes responsibility for the passed matrix specification. All of the passed data are assumed to be valid. The matrix specification may be `NULL`, in which case a matrix specification for a scalar is generated. If the argument `copy` is true, then the data will be copied. Otherwise, the memory pointed to by the `data` pointer will be used. If `data` is `NULL` enough memory to hold the matrix is allocated, regardless of the state of the `copy` variable.

Returns

It returns a pointer to a dynamically allocated and initialized structure. Upon error it returns `NULL` and sets `bpipe_errno`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPENOMEM`
: a memory allocation failed

A.5.49 hdrf_node_name_cmp

Compare two header fields using string collating sequence.

Synopsis

```
#include <bpipe/bpipe.h>

static int hdrf_node_name_cmp(
  const void *field1,
  const void *field2
);
```

Parameters

`const void *field1`
: a pointer to the first field

`const void *field2`
: a pointer to the second field

Description

Compare two header fields using string collating sequence.

Returns

It returns '`-1`', '`0`', or '`1`' if the first field's name is less than, equal to, or greater than, the second's, respectively.

A.5.50 matrix_memcpy

`memcpy` type process function for `bpipe_matrix_copy`

Synopsis

```
#include <bpipe/bpipe.h>

static int matrix_memcpy(
  void *udata,
  size_t dest,
  size_t src,
  size_t nbytes
);
```

Parameters

`void *udata`
: Not Documented.

`size_t dest`
: Not Documented.

`size_t src`
: Not Documented.

`size_t nbytes`
: Not Documented.

Description

This routine is a process routine for `bpipe_datatype_copy`. It does an immediate copy of the data chunk. The `udata` passed to it must be a character array with two elements. The first should point to the beginning of the destination data, the second should point to the beginning of the source data.

A.5.51 outchannel_close_delete

Close and delete an output channel.

Synopsis

```
#include <bpipe/bpipe.h>
```

```
static void outchannel_close_delete(void *node);
```

Parameters

`void *node`
: Not Documented.

Description

This is an action routine passed to the `linklist` package for the purpose of closing and deleting a list of output channels

A.5.52 read_dpkt_defs

Read data packet field definitions from a binary pipe.

Synopsis

```
#include <bpipe/bpipe.h>

static int read_dpkt_defs(
  BPipe *bpipe,
  char **buf,
  size_t *buf_len
);
```

Parameters

`BPipe *bpipe`
: the binary pipe to read

`char **buf`
: the input buffer to use

`size_t *buf_len`
: the length of the input buffer

Description

This routine reads in data packet field definitions from a binary pipe, parses them, and stores th results in the binary pipe structure. It requires a line count variable, which it increments and uses for error output. It uses `bpipe_proc_def` to parse the input. an empty line is taken to signal the end of the data packet field definition section.

Returns

It returns zero upon success, non-zero upon failure. It sets `bpipe_errno` upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADPIPE`
: the header data definitions had errors

`BPENOMEM` a memory allocation failed

Warning

The header parameter definitions must be read before this routine is called.

A.5.53 read_hdr_defs

Read and store header parameter definitions.

Synopsis

```
#include <bpipe/bpipe.h>

static int read_hdr_defs(
  BPipe *bpipe,
  char **buf,
  size_t *buf_len
);
```

Parameters

`BPipe *bpipe`
: the binary pipe to read

`char **buf`
: the input buffer to use

`size_t *buf_len`
: the length of the input buffer

Description

This routine reads header parameter definitions from a binary pipe, parses them, and stores the results in the binary pipe structure.

Returns

It returns zero upon success, non-zero upon failure. It sets `bpipe_errno` upon failure.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADPIPE`
: the header data definitions had errors

`BPENOMEM` a memory allocation failed

A.5.54 read_hdr

Read the header section of a binary pipe data stream.

Synopsis

```
#include <bpipe/bpipe.h>
```

```
static int read_hdr(BPipe *bpipe);
```

Parameters

`BPipe *bpipe`
: the binary pipe from which to read

Description

This routine reads and parses the header information of a binary pipe input data stream. The header consists of header parameter and data packets field definitions and header parameter data. The pipe I/O channel must already have been opened.

Returns

It returns zero upon success, non-zero otherwise. Upon error `bpipe_errno` is set.

Errors

Upon error `bpipe_errno` is set to one of the following errors:

`BPEBADPIPE`
: the header data definitions had errors

`BPEIOERROR`
: an error occurred whilst reading the pipe

`BPENOMEM` a memory allocation failed

A.5.55 write_defn

Write a field definition to an I/O channel.

Synopsis

```
#include <bpipe/bpipe.h>

static int write_defn(
  IOchannel *channel,
  char *name,
  BPDataType type,
  BPMatrix *matrix
);
```

Parameters

`IOchannel *channel`
: the iochannel

`char *name`
: the name of the data

`BPDataType type`
: the type of the data

Possible values for a BPDataType are as follows: BPDType_char, BPDType_double, BPDType_int, BPDType_uint, BPDType_DVector2, BPDType_DVector3, BPDType_IVector2, BPDType_IVector3, BPDType_UIVector2, BPDType_UIVector3, BPDType_DComplex, BPDType_DCVector2, BPDType_DCVector3

BPMatrix *matrix
: the matrix

Description

This routine writes a definition (either header field or data packet field) to an I/O channel. It's kludgy to avoid overrunning strings. one big sprintf would look nicer. maybe a bpipe_iochannel_printf?

Diagnostics

It returns zero upon success, non-zero upon error.

A.5.56 write_dpktf_def

Write the definition of a data packet field to an output channel

Synopsis

```
#include <bpipe/bpipe.h>

static int write_dpktf_def(
  void *data,
  void *udata
);
```

Parameters

void *data
: Not Documented.

void *udata
: Not Documented.

Description

This is an action routine used when traversing the list of data packet fields which writes out the field definitions.

A.5.57 write_hdrf_data

Write a header field's data to an output channel.

Synopsis

```
#include <bpipe/bpipe.h>

static int write_hdrf_data(void *data);
```

Parameters

`void *data`
: Not Documented.

Description

This is an action routine used when traversing the list of header fields which writes out the header field data.

Returns

It returns zero upon success, non-zero upon failure. It sets `bpipe_errno` to `BPEIOERROR` upon failure.

A.5.58 write_hdrf_def

Write the definition of a header field to an output channel

Synopsis

```
#include <bpipe/bpipe.h>

static int write_hdrf_def(
  void *data,
  void *udata
);
```

Parameters

`void *data`
: Not Documented.

`void *udata`
: Not Documented.

Description

This is an action routine used when traversing the list of header fields which writes out the field definitions.

A.5.59 xmap_compact

Compact a list of transformation maps

Synopsis

```
#include <bpipe/bpipe.h>

static Xmap *xmap_compact(size_t *nmap);
```

Parameters

`size_t *nmap`
: the number of elements in the resultant `Xmap` array

Description

`xmap_compact` operates on the file global linked list `xmap_ll`. This list consists of `Xmap` structures which specify the copy operations required to map between two data packet images. `xmap_compact` combines contigous copy operations. `xmap_ll` *must* contain at least one element.

`xmap_ll` is deleted after it is processed, even upon error.

Returns

On error it returns `NULL` and sets `bpipe_errno`, else it returns a pointer to an array of `Xmap` structures which define the transformations. It returns the length of the array via the parameter `nmap`.

Errors

Upon error `bpipe_errno` is set to one of the following:

`BPEBADARG`
: the list had no entries.

`BPENOMEM`
: a memory allocation failed

A.5.60 xmap_process

xmap_process

Synopsis

```
#include <bpipe/bpipe.h>

static int xmap_process(
  void *udata,
  size_t dst,
  size_t src,
  size_t size
);
```

Parameters

`void *udata`
: copy routine specific data

`size_t dst`
: destination of chunk

`size_t src`
: source of chunk

`size_t size`
: size of chunk in bytes

Description

This is a callback routine invoked by `bpipe_datatype_copy`. Instead of copying, it stores the source, destination, and length in a file global linked list, which will be compacted by `xmap_compact` after all data packet fields have been processed.

Returns

It returns zero upon success, non-zero upon failure (out of memoryb). It sets `bpipe_errno` to `BPENOMEM` upon failure.

Appendix B Examples

Here are a few example programs.

B.1 create.c

This program is a BPipe data stream source. It creates header fields and some data packet fields, and generates data packets.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <bpipe/bpipe.h>
#include <bpipe/datatypes.h>

/* simple print error and exit routine */
#define error(string) \
  do { \
      fprintf( stderr, __FILE__ " %d: %s\n", __LINE__, string ); \
      exit( EXIT_FAILURE ); \
     } while (0)

/* ------------------------------------------------------------------------*/

/* forward definitions */
void *dpktf_data( BPipe *bpipe, char *name, void *core_image,
  BPDataType type );

/* ------------------------------------------------------------------------*/

int
main (int argc, char *argv[])
{
  BPipe *bpipe;
  BPipeOutput *bpo;
  BPMatrix *matrix;
  void *core;

  unsigned int nspokes = 200;
  unsigned int nspokes_new = 13;
  unsigned int nrings = 200;

  DVector3 *position, *direction;
  double *polarization;
  double *graze_angle;
  double *weight;
```

```
int stuff_init;
size_t i;

size_t nphots = 1;

if ( argc > 1 )
  nphots = atoi( argv[1] ) || 1;

if ( NULL == ( bpipe = bpipe_new( ) ) )
  error( bpipe_strerror( bpipe_errno ) );

/* ------------------------------------------------------------ */

/*
   create new header fields. note that none of the following
   calls to bpipe_hdrf_add has the copy flag set, as there's no
   need to duplicate this data
 */

if ( bpipe_hdrf_add( bpipe, "nrings", BPDType_uint, NULL, &nrings, 0) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_hdrf_add( bpipe, "nspokes", BPDType_uint, NULL, &nspokes, 0) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_hdrf_string_add( bpipe, "comment", "first comment",
    (size_t) 0, (size_t) 0 ) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_hdrf_string_add( bpipe, "comment", "second comment",
    (size_t) 0, (size_t) 0 ) )
  error( bpipe_strerror( bpipe_errno ) );

/* ------------------------------------------------------------ */

/* create new data packet fields */

if ( bpipe_dpktf_add( bpipe, "position", BPDType_DVector3, NULL ) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_dpktf_add( bpipe, "direction", BPDType_DVector3, NULL ) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_dpktf_add( bpipe, "graze_angle", BPDType_double,   NULL ) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_dpktf_add( bpipe, "weight", BPDType_double,   NULL ) )
```

```
  error( bpipe_strerror( bpipe_errno ) );

/* create a one dimensional array with 4 elements */
if ( bpipe_dpktf_array_add( bpipe, "polarization", BPDType_double,
    (size_t) 4 ) )
  error( bpipe_strerror( bpipe_errno ) );

/*
   create a 3x3x3 cube. since the bpipe library takes responsibility for
   the matrix after a bpipe_dpktf_add, don't need to delete the matrix
 */
if ( NULL == (matrix = bpipe_matrix_new_va( (size_t) 3, 3, 3, 3 )) ||
     bpipe_dpktf_add( bpipe, "stuff", BPDType_int, matrix ) )
  error( bpipe_strerror( bpipe_errno ) );

/* ------------------------------------------------------------ */

/* resize nspokes array for the fun of it */

if ( NULL ==
     ( matrix = bpipe_hdrf_matrix( bpipe, "nspokes", BPHdrfIdx_LAST )) )
  error( bpipe_strerror( bpipe_errno ) );

matrix->extent[0] = 5;

/*
  initialize it with the contents of nspokes_new (set in the
  definitions above)
 */
if ( bpipe_hdrf_resize( bpipe, "nspokes", BPHdrfIdx_LAST, matrix,
     NULL, NULL, NULL, &nspokes_new, (size_t) 1 ) )
  error( bpipe_strerror( bpipe_errno ) );

/* ------------------------------------------------------------ */

/* open up output stream to stdout */
if ( NULL == ( bpo = bpipe_output( bpipe, "stdout" ) ) )
  error( bpipe_strerror( bpipe_errno ) );

/* map data packet fields */
if ( NULL == ( core = bpipe_map_alloc( bpipe, 1, NULL) ) &&
     bpipe_errno != BPNOERROR )
  error( bpipe_strerror( bpipe_errno ) );

/* output header. must do this after calling bpipe_map */
if ( bpipe_write_hdr(bpipe) )
  error( bpipe_strerror( bpipe_errno ) );
```

```
  /*
    get positions of fields in data packet. dpktf_data is *not* a
    bpipe library routine
   */
  position     = (DVector3 *) dpktf_data( bpipe, "position",     core, BPDType_DVector
  direction    = (DVector3 *) dpktf_data( bpipe, "direction",    core, BPDType_DVector
  weight       = (double *)   dpktf_data( bpipe, "weight",       core, BPDType_double
  graze_angle  = (double *)   dpktf_data( bpipe, "graze_angle",  core, BPDType_double
  polarization = (double *)   dpktf_data( bpipe, "polarization", core, BPDType_double

  /*
     use the fact that there's only one data packet core image buffer
     to preset some stuff
   */
  *weight = 235.3;
  *graze_angle = 1;

  for ( i = 0 ; i < 4 ; i++ )
    polarization[i] = i;

  stuff_init = 42;
  bpipe_dpktf_init( bpipe_dpktf( bpipe, "stuff" ), core, &stuff_init );

  /* create the data packets */
  for (i = 0; i < nphots ; i++)
  {
    /* play with the information */
    position->x = position->y = position->z = i;

    direction->x = direction->y = direction->z = i+2;

    /* write the data packet out */
    if ( bpipe_write_dpkt( bpipe, core, bpo ) )
      error( "error writing photon" );
  }

  bpipe_delete(bpipe);

  return EXIT_SUCCESS;
}

/* simple routine to get the data pointer for a named data packet field */
void *
dpktf_data( BPipe *bpipe, char *name, void *core_image, BPDataType type )
{
  void *data = bpipe_dpktf_datap( bpipe, core_image, name, type );
```

```
  if ( NULL == data )
    error( name );

  return data;
}
```

B.2 manip1.c

This program illustrates how to extend a data packet field's dimensions.

```
#include <stdlib.h>
#include <stdio.h>

#include <bpipe/bpipe.h>

/* simple print error and exit routine */
#define error(string) \
  do { \
      fprintf( stderr, __FILE__ " %d: %s\n", __LINE__, string ); \
      exit( EXIT_FAILURE ); \
     } while (0)

int
main (int argc, char *argv[])
{
  BPipe *bpipe;
  BPipeOutput *bpo;
  DpktField *dpktf;
  BPMatrix *matrix;
  void *data;
  size_t *dst_off;

  double init = argc > 1 ? atof(argv[1]) : 0.0;


  bpipe_errno = BPNOERROR;

  if ( NULL == ( bpipe = bpipe_new( ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( bpipe_input( bpipe, "stdin" ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( NULL == ( bpo = bpipe_output( bpipe, "stdout" ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /*
```

```
   resize the weight data field, increase the dimension by one. if
   there isn't a weight field, it's an error
 */
if ( NULL ==
     ( dpktf = bpipe_dpktf( bpipe, "weight" ) ) )
  error( bpipe_strerror( bpipe_errno ) );

if ( NULL == ( matrix = bpipe_dpktf_matrix( dpktf, BPDSite_CORE, NULL ) ) )
  error( bpipe_strerror( bpipe_errno ) );

/*
   make a copy of the field's matrix and give it an extra dimension with
   an extent of 2, essentially doubling the size of the matrix.
 */
matrix->nd++;
if ( NULL ==
      (matrix->extent = realloc( matrix->extent, matrix->nd * sizeof(size_t) )) )
  error( "couldn't realloc matrix" );
matrix->extent[matrix->nd - 1] = 2;

/*
   copy the data to the new matrix by moving everything to the other
   "half" of the doubled matrix
 */
if ( NULL == (dst_off = bpipe_offset_new( matrix->nd, (size_t) 0 ) ) )
  error( bpipe_strerror( bpipe_errno ) );

dst_off[matrix->nd - 1] = 1;

/*
   resize it.  leave most of the details to the routine. the bpipe library
   now has responsibility for the matrix and the offset array.
 */
if ( bpipe_dpktf_resize_core( dpktf, matrix, NULL, dst_off, NULL) )
  error( bpipe_strerror( bpipe_errno ) );

/* since there *is* a data packet field, need only check if the return
   sized is 0 and not bpipe_errno as well */
if ( NULL == ( data = ( bpipe_map_alloc( bpipe, 1, NULL ) ) ) )
  error( bpipe_strerror( bpipe_errno ) );

if ( bpipe_write_hdr(bpipe) )
  error( bpipe_strerror( bpipe_errno ) );

/*
   since the weight field is now extended, there will be uninitialized
   elements.  initialize the spot in the core image which holds
```

```
     the weight field.
   */

  bpipe_dpktf_init( dpktf, data, &init );

  /* loop through data packets.  not much to do here! */
  while(  bpipe_read_dpkts( bpipe, data, (size_t) 1 )   &&
         !bpipe_write_dpkt( bpipe, data, bpo )    )
    ;

  /*
    bpipe_errno should have been equal to BPNOERROR before the above
    loop
   */
  if ( bpipe_errno != BPNOERROR )
    error( bpipe_strerror( bpipe_errno ) );

  bpipe_delete(bpipe);

  return EXIT_SUCCESS;
}
```

B.3 create2.c

This program is a BPipe data stream source. It creates a data packet with a single two dimensinal matrix. Its output is designed to be used with manip4.c for good effect

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <bpipe/bpipe.h>

/* simple print error and exit routine */
#define error(string) \
  do { \
      fprintf( stderr, __FILE__ " %d: %s\n", __LINE__, string ); \
      exit( EXIT_FAILURE ); \
     } while (0)

int
main (int argc, char *argv[])
{
  BPipe *bpipe;
  BPipeOutput *bpo;
  DpktField *dpktf;
```

```
  BPMatrix *matrix;
  void *data;

  double init = 1.0;
  size_t i;

  size_t nphots = 1;

  if ( argc > 1 )
    nphots = atoi( argv[1] );

  if ( NULL == ( bpipe = bpipe_new( ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  matrix = bpipe_matrix_new_va( (size_t) 2, 1, 1 );
  /* create new data packet fields */
  if ( bpipe_dpktf_add( bpipe, "weight", BPDType_double,   matrix ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( NULL == ( bpo = bpipe_output( bpipe, "stdout" ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /* map data packet fields */
  if ( NULL == ( data = bpipe_map_alloc( bpipe, 1, NULL ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /* output header */
  if ( bpipe_write_hdr(bpipe) )
    error( bpipe_strerror( bpipe_errno ) );

  /* get handles to data packet fields */
  dpktf = bpipe_dpktf( bpipe, "weight" );

  bpipe_dpktf_init( dpktf, data, &init );

  /* write the data packets out */
  for (i = 0; i < nphots ; i++)
    if ( bpipe_write_dpkt( bpipe, data, bpo ) )
      error( "error writing photon" );

  bpipe_delete(bpipe);

  return EXIT_SUCCESS;
}
```

B.4 manip4.c

This program takes a data packet field and embeds it in a larger matrix such that there's one element along each side of the old data. It works well with `create2.c`.

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <bpipe/bpipe.h>

/* simple print error and exit routine */
#define error(string) \
  do { \
      fprintf( stderr, __FILE__ " %d: %s\n", __LINE__, string ); \
      exit( EXIT_FAILURE ); \
     } while (0)

int
main (int argc, char *argv[])
{
  BPipe *bpipe;
  BPipeOutput *bpo;
  DpktField *dpktf;
  BPMatrix *matrix;
  void *data;
  double init = argc > 1 ? atof(argv[1]) : 0.0;
  size_t *dst_off;

  size_t i;

  if ( NULL == ( bpipe = bpipe_new( ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( bpipe_input( bpipe, "stdin" ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( NULL == ( bpo = bpipe_output( bpipe, "stdout" ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /* resize the weight data field and move the current data, such that
     we create a single element border of new elements along all sides
     of the data */
  if ( NULL == ( dpktf = bpipe_dpktf( bpipe, "weight" ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( NULL == ( matrix = bpipe_dpktf_matrix( dpktf, BPDSite_CORE, NULL ) ) )
```

```
    error( bpipe_strerror( bpipe_errno ) );

  /* tweak the matrix */
  for ( i = 0 ; i < matrix->nd ; i++ )
    matrix->extent[i]+=2;

  if ( NULL == ( dst_off = bpipe_offset_new( matrix->nd, (size_t) 1 ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /* resize it.  leave most of the details to the routine */
  if ( bpipe_dpktf_resize_core( dpktf, matrix, NULL, dst_off, NULL) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( NULL == ( data = bpipe_map_alloc( bpipe, 1, NULL ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  if ( bpipe_write_hdr(bpipe) )
    error( bpipe_strerror( bpipe_errno ) );

  /* since the weight field is now extended, there will be uninitialized
     elements.  initialize the spot in the core image which holds
     the weight field */
  bpipe_dpktf_init( dpktf, data, &init );

  /* loop through data packets */
  bpipe_errno = BPNOERROR;
  while( bpipe_read_dpkts( bpipe, data, (size_t) 1 ) )
    if ( bpipe_write_dpkt( bpipe, data, bpo ) )
      error( "error writing photon" );

  if ( bpipe_errno != BPNOERROR )
    error( bpipe_strerror( bpipe_errno ) );

  bpipe_delete(bpipe);

  return EXIT_SUCCESS;
}
```

B.5 bpipe_dump.c

This program will dump the contents of an input `BPipe` stream (on `stdin`) to `stdout`, formatting it for human comprehension.

```
/* --8<--8<--8<--8<--
 *
 * Copyright (C) 2005-2011 Smithsonian Astrophysical Observatory
 *
 * This file is part of bpipe
```

```
 *
 * bpipe is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 * -->8-->8-->8-->8-- */

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <bpipe/bpipe.h>
#include <bpipe/datatypes.h>

/* simple print error and exit routine */
#define error(string) \
  do { \
      fprintf( stderr, __FILE__ " %d: %s\n", __LINE__, string ); \
      exit( EXIT_FAILURE ); \
     } while (0)

/* ------------------------------------------------------------------------*/

/* forward definitions */
void  print_matrix( BPMatrix *matrix, char *buf );
void * print_data( char *prefix, BPDataType type, BPMatrix *matrix,
                          void *data, size_t level );

/* ------------------------------------------------------------------------*/


int
main (int argc, char *argv[])
{
  BPipe *bpipe;
  DpktField *dpktf;
  BPMatrix *matrix;
```

```
void *data;
size_t index;
void *last;
const char *name;

size_t dpkt_core_size, n;

/* create BPipe object */
if ( NULL == ( bpipe = bpipe_new( ) ) )
  error( bpipe_strerror( bpipe_errno ) );

/* attach standard input to the BPipe */
if ( bpipe_input( bpipe, "stdin" ) )
  error( bpipe_strerror( bpipe_errno ) );

/* step through header fields and print out their definitions and data */
printf( "header fields:\n" );

/* last is used by bpipe_hdrf_next to keep track of where it is */
last = NULL; /* initialize it for first time through */

/* loop until there are no more header fields available */
while ( bpipe_hdrf_next( bpipe, &name, &index, &last ) )
{
  char buf[1024];

  /*
    extract the data type and a pointer to the data.  these should
    be valid since we've been handed name and index by bpipe_hdrf_next
   */
  BPDataType type = bpipe_hdrf_type( bpipe, name, index );
  void *     data = bpipe_hdrf_data( bpipe, name, index );

  if ( bpipe_errno != BPNOERROR )
    error( bpipe_strerror( bpipe_errno ) );

  /*
    extract matrix spec.  since it's copying the matrix, there's a
    possibility that we're out of memory, which we check for
   */
  if ( NULL == (matrix = bpipe_hdrf_matrix ( bpipe, name, index ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /*
    generate the field definition.  this is used to prefix the data
    output
   */
```

```
  sprintf(buf, "%s-%lu", name, (unsigned long) index );
  print_matrix( matrix, buf );
  strcat( buf, ": " );

  /* dump the data */

  /*
    if it's a one dimensional character array, assume that it's a C
    string
   */
  if ( BPDType_char == type && matrix->nd == 1 )
    printf( "%s%s\n", buf, (char*) data );
  else
    print_data( buf, type, matrix, data, matrix->nd - 1 );

  /*
    we're responsible for deleting the matrix spec returned by
    bpipe_hdrf_matrix
   */
  bpipe_matrix_delete( matrix );
}

/* now print out the data packet field definitions */

printf( "\ndata fields:\n" );
last = NULL; /* initialize it for first time through */

while (( dpktf = bpipe_dpktf_next( bpipe, &last ) ))
{
  char buf[1024];

  /*
    extract matrix spec.  since it's copying the matrix, there's a
    possibility that we're out of memory, which we check for
   */
  if ( NULL == (matrix = bpipe_dpktf_matrix( dpktf, BPDSite_CORE, NULL ) ) )
    error( bpipe_strerror( bpipe_errno ) );

  /* print out the definition */
  strcpy( buf, bpipe_dpktf_name( dpktf ) );
  print_matrix( matrix, buf );
  puts( buf );

  /*
    we're responsible for deleting the matrix spec returned by
    bpipe_dpktf_matrix
   */
```

```
    bpipe_matrix_delete( matrix );
  }

  /*
    we haven't manipulated any fields, but still must map the data
    packet images to get the size of the core image
   */

  /*
    since we haven't done anything, bpipe_map should only fail if it
    runs out of memory.  note that its possible that there are no data
    packet fields, in which case a return of zero isn't an error
   */

  if ( 0 == ( dpkt_core_size = bpipe_map( bpipe ) ) &&
       bpipe_errno != BPNOERROR )
    error( bpipe_strerror( bpipe_errno ) );

  /* if there are no data packets, there's not much left to do */
  if ( dpkt_core_size == 0 )
  {
    bpipe_delete( bpipe );
    return EXIT_SUCCESS;
  }

  /* space for instance of data packet */
  if ( NULL == (data = malloc( dpkt_core_size )) )
    error( "unable to allocate data packet" );

  /* loop through data packets */
  n = 0;
  while( bpipe_read_dpkts( bpipe, data, (size_t) 1 ) )
  {
    n++;
    printf( "\ndata packet %lu:\n", (unsigned long) n );

    last = NULL;

    while (( dpktf = bpipe_dpktf_next( bpipe, &last ) ))
    {
      char buf[1024];

      /*
for speed, we could have saved these while printing the field
definitions above, but these are pretty cheap calls
       */
      void *dpktf_data = bpipe_dpktf_data( dpktf, data );
```

```
      BPDataType  type = bpipe_dpktf_type( dpktf );

      if ( bpipe_errno != BPNOERROR )
        error( bpipe_strerror( bpipe_errno ) );

      /*
this call is expensive, as it must copy the matrix.
ordinarily one would do this outside of the read loop and
store the values
       */
      if ( NULL ==
   (matrix = bpipe_dpktf_matrix( dpktf, BPDSite_CORE, NULL ) ) )
error( bpipe_strerror( bpipe_errno ) );

      strcpy( buf, bpipe_dpktf_name( dpktf ) );
      print_matrix( matrix, buf );
      strcat( buf, ": " );

      if ( BPDType_char == type && matrix->nd == 1 )
printf( "%s%s\n", buf, (char*) dpktf_data );
      else
print_data( buf, type, matrix, dpktf_data, matrix->nd - 1 );

      /*
we're responsible for deleting the matrix spec returned by
bpipe_dpktf_matrix
       */
      bpipe_matrix_delete( matrix );
    }
  }

  /*
    we'll get an error out of bpipe_read_dpkts if something's gone awry.
    bpipe_errno should have been equal to BPNOERROR when entering the
    above loop
   */
  if ( bpipe_errno != BPNOERROR )
    error( bpipe_strerror( bpipe_errno ) );

  bpipe_delete(bpipe);

  free( data );

  return EXIT_SUCCESS;
}

/* print out matrix spec */
```

```
void
print_matrix( BPMatrix *matrix, char *buf )
{
  int i;

  for ( i = 0 ; i < matrix->nd ; i++ )
    sprintf(buf + strlen( buf ),  "[%lu]", (unsigned long) matrix->extent[i] );
}

/* recursive N-dimensional matrix data print routine */
void *
print_data( char *prefix,
    BPDataType type, BPMatrix *matrix, void *data, size_t level )
{
  char buf[1024];
  size_t i;
  size_t size = bpipe_datatype_size( type );

  if (level == 0)
  {
    fputs( prefix, stdout );
    fputs( ": ", stdout );
    for (i = 0 ; i < matrix->extent[level] ; i++, data = (char *) data + size)
    {
      bpipe_sprintf( buf, data, type, NULL );
      fputs( buf, stdout );
      if ( i < matrix->extent[level] - 1 )
fputs( " | ", stdout );
    }
    putchar('\n');
    return data;
  }
  else
  {
    for ( i = 0 ; i < matrix->extent[level] ; i++ )
    {
      sprintf( buf, "%s[%lu]", prefix, (unsigned long) i );
      data = print_data( buf, type, matrix, data, level - 1 );
    }
    if ( level == 1 )
      puts( "--------------------" );
  }

  return data;
}
```