

# libaffy Reference Manual

---

covering version 1.3

Steven Eschrich  
Andrew Hoerter

---

This reference manual documents the 'libaffy' library, version 1.3.  
Copyright © 2006 H. Lee Moffitt Cancer Center & Research Institute

# Table of Contents

|          |                                       |           |
|----------|---------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                   | <b>1</b>  |
| <b>2</b> | <b>Building and Using the Library</b> | <b>2</b>  |
| <b>3</b> | <b>Common Tasks</b>                   | <b>3</b>  |
| 3.1      | DAT Images                            | 3         |
| 3.2      | CEL Files                             | 3         |
| 3.3      | CDF File                              | 3         |
| 3.4      | Processing CEL Files                  | 4         |
| <b>4</b> | <b>Data Types</b>                     | <b>5</b>  |
| 4.1      | Constants                             | 5         |
| 4.2      | Base Types                            | 5         |
| 4.3      | Data Structures                       | 5         |
| 4.3.1    | AFFY_PIXREGION                        | 5         |
| 4.3.2    | AFFY_POINT                            | 5         |
| 4.3.3    | AFFY_CELL                             | 6         |
| 4.3.4    | AFFY_PROBE                            | 6         |
| 4.3.5    | AFFY_PROBESET                         | 6         |
| 4.3.6    | AFFY_CDFFILE                          | 6         |
| 4.3.7    | AFFY_CELFILE                          | 7         |
| 4.3.8    | AFFY_DATFILE                          | 7         |
| 4.3.9    | AFFY_CHIP                             | 8         |
| 4.3.10   | AFFY_CHIPSET                          | 8         |
| 4.4      | Flag Structures                       | 9         |
| 4.4.1    | MAS5.0 Flags                          | 9         |
| 4.4.2    | RMA Flags                             | 10        |
| <b>5</b> | <b>Function Reference</b>             | <b>12</b> |
| 5.1      | File Access Functions                 | 12        |
| 5.2      | Statistical Functions                 | 15        |
| 5.3      | Utility Functions                     | 18        |
|          | <b>Concept Index</b>                  | <b>20</b> |

# 1 Introduction

'libaffy' provides various helpful functions for dealing with files containing Affymetrix (R) GeneChip (R) microarray data. It provides a simple interface for performing a variety of tasks:

- Loading data from 'CEL', 'CDF', and 'DAT' files, either individually or at the chip level (both V3 and V4 format)
- Representing this data in a convenient form
- Perform standard operations, including image rotation, RMA and MAS5.0
- Extensible codebase for developing Affymetrix chip-based applications.

The goal of this library is a reusable set of routines that could be integrated into different software. Since the code is written in C, a reasonable effort has been made to ensure the code is memory-efficient and as minimal as possible. As more and more microarrays are generated and the individual chip size increases, these issues will become increasingly important.

The library can also be integrated into the bioconductor package ([www.bioconductor.org](http://www.bioconductor.org)) thereby providing the best of both options.

## 2 Building and Using the Library

With a few minor exceptions, the ‘libaffy’ source code is ANSI C89 compliant, although it will take advantage of certain C99 and GNU C capabilities when they are present in the environment. If you wish to use the provided build system, you must install GNU (GNU’s Not Unix) `make` (please make sure to use the latest version; some older versions have bugs which prevent proper interpretation of the makefiles). The build system generally assumes a POSIX-like environment with basic utilities such as `rm`, `cp`, and so on. When building under Microsoft(R) Windows(R), an emulation package such as Cygwin or MinGW will be necessary to provide these POSIX programs. In the future build definition files for other environments may become available to eliminate this requirement.

The end result of the build process will be a library file, ‘libaffy.a’, which can then be linked against application code. The default target of the top-level ‘Makefile’ should produce the library.

## 3 Common Tasks

### 3.1 DAT Images

The Affymetrix DAT image file is a simple structure consisting of a 512-byte header followed by pixel data. A single function is used to load a named DAT file into the `AFFY_DATFILE` structure ( See [Section 5.1 \[File Access Functions\]](#), page 12, `affy_load_dat_file`). All pixels are currently loaded which can require significant amounts of memory for multiple chips. The in-memory structure is a series of fields that are listed within the header of the image, followed by a `AFFY_PIXREGION` object. The `AFFY_PIXREGION` structure is a convenient mechanism to store all or a subset of the image within a consistent structure.

Once data is loaded, you can extract various subsection of the image by cell index (e.g. the probe location on the chip within the CEL file). An extremely important accessor function for this access is `affy_pixels_from_cell`, which is used to extract an `AFFY_PIXREGION` from the image based on CEL coordinates. Note, this is non-trivial since the image is generally slightly rotated and coordinates must be translated into the rotated space before determining the appropriate pixels. The function returns an `AFFY_PIXREGION` which can then be used to calculate a single summary (which is what Affymetrix does).

Given an `AFFY_PIXREGION`, you may want to write out the data in an alternate format. The image (or a subset of the image as defined by an `AFFY_PIXREGION`) can be written out as text or a TIF image (see `affy_write_pixel_region` for details).

### 3.2 CEL Files

Typically Affymetrix microarray data is considered raw in the form of CEL files, despite the fact that the image is actually the most raw file format. Therefore, a central part of this library is the routine to read in a CEL file and the structure that stores a CEL file. Note: Affymetrix has developed a new version of their CEL file that is binary form (v4). In `libaffy`, the appropriate version is automatically detected and the appropriate code is called.

The central function to use in loading a CEL file is `affy_load_cel_file`, which returns an `AFFY_CELFILE` structure. This structure holds raw data (as a matrix) and does not do any translation of the coordinates to probes. In addition, it stores the cell locations marked as outliers or masked within the CEL file. These are bit strings, therefore there are accessor functions (`iscontrol` and `ismasked`) to probe these values without knowing how to manipulate the bit strings.

### 3.3 CDF File

Since the CEL file only consists of matrix of values (intensity, standard deviation and number of pixels), the CEL file is not useful without the corresponding CDF file. In order to load a CDF file, you use the `affy_load_cdf_file` routine. This function requires a chip type string, as defined by the basename of the CDF file (or equivalently the chip name within the CEL file) and a directory to search for the file in. The load routine will also look in the current directory for the file before giving up. Note that binary CDF file support is also automatically detected and used.

The chip type may be inconvenient to always specify within an application, therefore a utility function (`affy_get_cdf_name_from_cel`) can be used to extract the appropriate chip type from within a CEL file (for instance, the first CEL file in a list accepted from the user).

The CDF structure is the most complex, as is understandable given its importance to processing the microarray data. The structure contains `AFFY_PROBE` structures that combine to form `AFFY_PROBESET` structures. Both the probe and probeset information is stored, so that access can occur in whichever mode is most appropriate. Specifically, `AFFY_PROBE` structures are accessible via (X,Y) coordinates or as a linear array of probes. A linear array of probesets can also be accessed (which in turn point to probes). Finally, a matrix of cell type descriptors are included so that normal, undefined, control and QC cells can be identified easily.

The `rma` and `mas5` code consists of many good examples of the use of the CDF structures.

### 3.4 Processing CEL Files

Generally of most interest when loading a CEL file is converting the individual probe values to actual probeset expressions. The `libaffy` contains code for both the MAS5.0 and RMA algorithms. The details of computing MAS5.0 and RMA are not described here, but many good descriptions do exist.

The entry point for both MAS 5.0 and RMA consists of a single routine that is provided with an array of filenames and a flag structure. The filenames are assumed to be CEL files of the same type that are opened sequentially and processed. A meta-structure (`AFFY_CHIPSET`) is returned that represents the in-memory image of probeset expression and the corresponding CDF file. CEL file data is freed as chips are processed so memory usage is kept to a minimum. The flag structure is used to control the various parameters of the algorithm and are documented below and within the header files.

There is nothing overly complex about the `rma` or `mas5` functions. These functions merely call the various routines that actually implement the algorithms. Therefore a good place to experiment is within this function, for instance by removing background correction as a processing step.

## 4 Data Types

### 4.1 Constants

There are several constants defined within `libaffy`.

- `AFFY_DAT_FILEMAGIC` — Magic number for DAT files.
- `AFFY_CDF_BINARYFILEMAGIC` — Magic number for new CDF format.
- `AFFY_CEL_BINARYFILE_MAGIC` — Magic number for new CEL format.

There are also location types, which are used in a CDF file as the cell type:

- `AFFY_UNDEFINED_LOCATION` — Location is not specified within CDF.
- `AFFY_QC_LOCATION` — Location designated QC (quality control) by CDF.
- `AFFY_NORMAL_LOCATION` — Normal intensity location in CDF.

### 4.2 Base Types

The `libaffy` library is generally platform-independent, despite the fact that the Affymetrix data files are very platform-specific. Therefore several different base types are defined that have known size on each platform regardless of potential differences in `sizeof(int)` for example.

- `affy_uint8` is an unsigned byte.
- `affy_int16` is a signed 16-bit integer.
- `affy_uint16` is an unsigned 16-bit integer.
- `affy_int32` is a signed 32-bit integer.
- `affy_uint32` is an unsigned 32-bit integer.

### 4.3 Data Structures

#### 4.3.1 `AFFY_PIXREGION`

A region of pixels with the dimensions. Allows for efficient subsetting by allocating minimal data to point at a particular spot of a larger image.

```
affy_uint16 numrows;
affy_uint16 numcols;
unsigned int **data;          /* Must hold at least 16 bits; C89 says so. */
```

#### 4.3.2 `AFFY_POINT`

A simple structure to hold (x,y) coordinates together.

```
affy_int16 x;
affy_int16 y;
```

### 4.3.3 AFFY\_CELL

A cell is a location defined by the `CEL` file. The cell consists of a summary value, the standard deviation in the measurement and the number of pixels involved in the calculation (all from the `CEL` file). Can also indicate a number of pixels from the `DAT` file if the `DAT` file is loaded.

```
double value;           /* Intensity (mean pixel intensity). */
double stddev;        /* Std. deviation of pixel intensity. */
int numpixels;        /* Number of pixels composing this cell. */
AFFY_PIXREGION *pixels; /* The actual pixels making up the cell. */
```

### 4.3.4 AFFY\_PROBE

A probe is the basic unit of meta-information, consisting of both a perfect match point and a mismatch point (`x,y` coordinates). It also contains a pointer back to the probeset it belongs to.

```
int index;             /* Unique id for probe */
AFFY_POINT mm;        /* Location of Mismatch value */
AFFY_POINT pm;        /* Location of Perfect match value */
struct affy_probeset_s *ps; /* Pointer back to parent probe set */
```

### 4.3.5 AFFY\_PROBESET

The probeset consists of a set of individual probes. Eventually the probeset is summarized to a single value elsewhere, as a combination of multiple PM and MM probes.

```
int index;             /* Unique id for probe set */
char *name;           /* A text description of probe set */
int numprobes;        /* Total number of probes */
AFFY_PROBE *probe;    /* The probes themselves */
```

### 4.3.6 AFFY\_CDFFILE

The CDF file holds all information about how probes are put together to make probesets, as well as what every cell location is. This information is represented in memory as a bunch of connections to other structures.

Note that in this structure, the information is represented several different ways. First, a matrix of `cell_type` defines for each cell location what type of cell it is (normal, control, QC). Next, probes can be extracted by (`x,y`) coordinates directly. Or the probes can be accessed linearly (via `probe`) to iterate through all probes. Finally, all probesets can be iterated through — which ultimately links back to individual probes. These different access methods makes it convenient to do many different types of processing with the same structure.

```
char *array_type;     /* Name of file/chi
int numrows;
int numcols;
int numprobes;        /* Count of total probes */
int numprobesets;     /* Number of probe sets */
int numqcunits;      /* Num quality control units */
```

```

    affy_uint8 **cell_type;      /* What kind of cell (normal, qc, control) */
    AFFY_PROBE ***xy_ref;      /* row,col map to individual probe */
    AFFY_PROBESET *probeset;   /* An array of probe sets, made of probes */
    AFFY_PROBE **probe;        /* A linear array of probes
*/

```

### 4.3.7 AFFY\_CELFILE

The CEL file contains all of the actual intensity information for a single microarray. These intensities are simply listed indexed by their coordinates on the microarray. Therefore, this structure is relatively simple.

One noteworthy exception is the `mask` and `outlier` components. These are both bit strings that indicate for an  $(x,y)$  coordinate whether or not the cell location was marked as an outlier or masked. There are convenience functions `affy_iscontrol()`, `affy_isoutlier()` and `affy_ismasked()` that allow you to probe these values without knowing how they are implemented.

```

    char *filename;
    int numrows;
    int numcols;
    int nummasks;
    int numoutliers;
    AFFY_CELL **data;
    affy_uint8 **mask;
    affy_uint8 **outlier;

```

### 4.3.8 AFFY\_DATFILE

The DAT file is the original image of the microarray chip. The header contains a variety of fields that are not generally useful but captured in memory nonetheless. The geometry of the image and the pixels themselves are conveniently represented within the `pixels` variable, which is an `AFFY_PIXREGION` and is thus self-contained.

The grid corners are noted within the DAT image and are important for calculating the rotation of the image. This rotation can be calculated by using the `affy_pixels_from_cell()` or `affy_cell_to_pixel()` routines to translate between cell coordinates and actual pixels without having to know any further details of the algorithm.

```

    AFFY_PIXREGION pixels;
    affy_uint32 numpixels;
    AFFY_POINT grid_ul;
    AFFY_POINT grid_ur;
    AFFY_POINT grid_ll;
    AFFY_POINT grid_lr;

    char *experiment_name;
    affy_uint16 scanspeed;
    affy_float32 temperature;
    affy_float32 laser_power;
    char timestamp[19];          /* field width 18, plus NUL */
    affy_uint32 numsamples_dc_offset;

```

```

affy_uint16 cellmargin;
char *scannerid;
char *probe_array_type;
affy_float64 meanpixel;
affy_float64 std_dev_pixel;
affy_float64 avg_dc_offset;
affy_float64 std_dev_dc_offset;
affy_uint16 pixel_width;
affy_uint16 pixel_height;
affy_uint32 minpixel;
affy_uint32 maxpixel;

```

### 4.3.9 AFFY\_CHIP

A higher abstraction of a microarray experiment is the chip. Here, we store pointers to the various raw data sources (CDF,CEL,DAT) as well as the final result — an array of probeset expression values.

```

char *filename;          /* Filename of chip          */
AFFY_CDFFILE *cdf;      /* The chip description file */
AFFY_CELFILE *cel;      /* For the CEL file information */
AFFY_DATFILE *dat;      /* Underlying pixel data (if present) */

/* The true information - the probe sets */
int numprobesets;
double *probe_set;

/* A convenience ptr: not globally used (RMA) */
double *pm;

```

### 4.3.10 AFFY\_CHIPSET

A chipset holds multiple chips of the same type. There is an array of chips and a pointer to a CDF that describes all of the chips.

```

/* Some common parameters needed, from the CDF file. */
int numchips;
int numrows;
int numcols;

/* Each CHIPSET has a single description file (CDF) */
AFFY_CDFFILE *cdf;

/*
 * These are the chips themselves. Note they are arrays of ptrs,
 * so things can be copied more easily
 */
AFFY_CHIP **chip;

```

## 4.4 Flag Structures

### 4.4.1 MAS5.0 Flags

MAS5\_FLAGS is defined as follows (can be seen in 'affy\_mas5.h').

```

/**
 * Structure to hold all relevant information
 * about running an instance of MAS5.0. The
 * defaults for these fields are in parenthesis, but
 * the ultimate authority for these values is defined
 * within the file mas5/mas5_set_defaults.c.
 */

/** (.) Default location to look for CDF file */
char *cdf_directory;

/** (True) Run MAS5.0 background correction */
Boolean use_background_correction;

/** (False) Use a mean normalization of all probes prior to processing */
Boolean use_mean_normalization;
/** (500) Mean normalize all probes to target mean value */
double mean_normalization_target_mean;

/** (False) Use a quantile normalization of all probes prior to processing.
 *      This option is mutually exclusive with use_mean_normalization.
 */
Boolean use_quantile_normalization;

/** (True) Scale probesets to a constant value, determined as scale_target */
Boolean use_probeset_scaling;
/** (500) If probeset scaling is used, this should be the target trimmed mean */
double scale_target;
/** (0.02) Percentage below which to remove from trimmed mean */
double trimmed_mean_low;
/** (0.98) Percentage above which to remove from trimmed mean */
double trimmed_mean_high;

/** (false) Use Bioconductor compatability mode */
Boolean bioconductor_compatability;

/** (16) The number of rectangular zones on the chip */
int K;
/** (100) MAS5.0 smooth parameter */
int smooth;

```

```

/** (0.5) MAS5.0 noiseFrac parameter */
double NoiseFrac;
/** (2.0e-20) MAS5.0 delta parameter */
double delta;
/** (0.03) MAS5.0 Contrast tau parameter */
double contrast_tau;
/** (10) MAS5.0 Scale tau parameter */
double scale_tau;

```

#### 4.4.2 RMA Flags

RMA\_FLAGS is defined as follows (can be seen in 'affy\_rma.h').

```

/**
 * Structure to hold all relevant information
 * about running an instance of RMA. The defaults
 * for these fields are in parenthesis, but the
 * ultimate authority for these values is defined
 * within the file rma/rma_set_defaults.c.
 */
typedef struct rma_flag_struct
{
    /** (.) Default location to look for CDF file */
    char *cdf_directory;

    /** (True) Run background correction */
    Boolean use_background_correction;

    /** (True) Run normalization step */
    Boolean use_normalization;

    /** (True) Use AFFX (control) probes when normalizing */
    Boolean normalize_affx_probes;

    /** (False) Use mean normalization instead of quantile normalization */
    Boolean use_mean_normalization;

    /** (500) If using mean normalization, set mean equal to target */
    double mean_normalization_target_value;

    /** (False) Dump probe affinities to a file */
    Boolean dump_probe_affinities;

    /** ("affinities.txt") Optional affinity filename */
    char *affinities_filename;
}

```

```
} RMA_FLAGS;
```

## 5 Function Reference

### 5.1 File Access Functions

**AFFY\_CDFFILE** *\*affy\_load\_cdf\_file (char \*chip\_type, char \*dir)* [File Access]

Given a chip type and directory (generally, the chip type corresponds to a name, while the directory is a platform-specific directory name), attempt to load the specified ‘CDF’ file into memory. The version of CDF file is automatically detected.

If the load is successful, a new **AFFY\_CDFFILE** structure is filled out and a pointer to it is returned. Otherwise, the library will end execution with a fatal error.

It is the caller’s responsibility to free the returned structure using **affy\_free\_cdf\_file()**. However, only structures returned by direct allocation should be freed; those which were packaged inside another dynamically allocated structure will be handled when the containing structure is freed, and do not need to be explicitly freed.

**void** *affy\_free\_cdf\_file (AFFY\_CDFFILE \*cdf)* [File Access]

Given a valid **AFFY\_CDFFILE** structure, the associated storage is freed.

It is an error to access the structure pointed to by **cdf**, or any of its members, after calling this function.

**AFFY\_CELFILE** *\*affy\_load\_cel\_file (char \*filename)* [File Access]

Given a platform-dependent filename attempt to load the specified ‘CEL’ file into memory.

If the load is successful, a new **AFFY\_CELFILE** structure is filled out and a pointer to it is returned. Otherwise, the library will end execution with a fatal error.

It is the caller’s responsibility to free the returned structure using **affy\_free\_cel\_file()**. However, only structures returned by direct allocation should be freed; those which were packaged inside another dynamically allocated structure will be handled when the containing structure is freed, and do not need to be explicitly freed.

**void** *affy\_free\_cel\_file (AFFY\_CELFILE \*cf)* [File Access]

Given a valid **AFFY\_CELFILE** structure, the associated storage is freed.

It is an error to access the structure pointed to by **cf**, or any of its members, after calling this function.

**AFFY\_DATFILE** *\*affy\_load\_dat\_file (char \*filename)* [File Access]

Given a platform-specific filename attempt to load the specified ‘DAT’ file into memory. All header fields and all pixel intensities are loaded in memory.

If the load is successful, a new **AFFY\_DATFILE** structure is filled out and a pointer to it is returned. Otherwise, the library will end execution with a fatal error.

It is the caller’s responsibility to free the returned structure using **affy\_free\_dat\_file()**. However, only structures returned by direct allocation should be freed; those which were packaged inside another dynamically allocated structure will be handled when the containing structure is freed, and do not need to be explicitly freed.

**void** *affy\_free\_dat\_file* (*AFFY\_DATFILE \*df*) [File Access]

Given a valid *AFFY\_DATFILE* structure, the associated storage is freed.

It is an error to access the structure pointed to by *df*, or any of its members, after calling this function.

*AFFY\_CHIPSET \*affy\_load\_data* (*char \*dir, char \*\*files*) [File Access]

The data files comprising one or more chips listed in directory *dir* and specified by *files* are loaded. Either of *dir* or *files* can be set to NULL (but not both).

The ‘CEL’ files named by *files* are opened, as is the corresponding ‘CDF’ file. The resulting *AFFY\_CHIPSET* is returned as a dynamically allocated object.

It is the caller’s responsibility to free the returned structure using *affy\_free\_chipset*() .

**void** *affy\_free\_chipset* (*AFFY\_CHIPSET \*cs*) [File Access]

Given a valid *AFFY\_CHIPSET* structure, the associated storage is freed.

It is an error to access the structure pointed to by *cs*, or any of its members, after calling this function.

**void** *affy\_free\_chip* (*AFFY\_CHIP \*ch*) [File Access]

Given a valid *AFFY\_CHIP* structure, the associated storage is freed.

It is an error to access the structure pointed to by *ch*, or any of its members, after calling this function.

*AFFY\_CHIP \*affy\_load\_chip* (*char \*filename*) [File Access]

Initialize a *AFFY\_CHIP* structure with the CEL data assumed to be in *filename*. No other data is loaded. A NULL pointer is returned on error.

**char \*\****affy\_list\_files* (*char \*directory, char \*extension*) [File Access]

Return a list of files within *directory* with the file extension *extension*. The return value is an array of filenames with a NULL as the last element (*argv* style). An empty list returns an array with one element, a NULL.

**char \****affy\_get\_next\_line* (*FILE \*fp*) [File Access]

Read a single line of text from the open filestream *fp*, trim any whitespace from both ends, and return the result. NULL is returned on error or in the case of an empty string.

The resulting pointer refers to static storage and should not be used across invocations of *affy\_get\_next\_line*() .

**void** *affy\_unget\_next\_line* (*FILE \*fp*) [File Access]

Causes the next invocation of *affy\_get\_next\_line*() to return the current contents of the internal line buffer without reading another line from filestream *fp*.

**void** *affy\_reset\_next\_line* (*FILE \*fp*) [File Access]

This function has the opposite effect from *affy\_unget\_next\_line*(); the next invocation of *affy\_get\_next\_line*() will return the next line from filestream *fp* rather than the current contents of the internal line buffer.

- void** *affy\_skip\_to\_next\_header* (*FILE \*fp*) [File Access]  
Lines will be read from filestream *fp* until the first non-whitespace character of a line is a left bracket ('['). Any such line encountered will be the next line returned by *affy\_get\_next\_line()*.
- int** *affy\_read8* (*FILE \*input*, *void \*buf*) [File Access]  
Given an open input filestream *input* and a valid storage location pointed to by *buf*, one byte will be read and stored in *buf*.  
On success, '0' is returned. On failure, '-1' is returned and the state of the target buffer is undefined. The caller is responsible for clearing or handling errors on the filestream.
- int** *affy\_read16* (*FILE \*input*, *void \*buf*) [File Access]  
Given an open input filestream *input* and a valid storage location pointed to by *buf*, two bytes will be read and stored in *buf*. The input stream is assumed to be in little-endian format; if the host platform is big-endian, the bytes will be swapped automatically.  
On success, '0' is returned. On failure, '-1' is returned and the contents of the target buffer are undefined. The caller is responsible for clearing or handling errors on the filestream.
- int** *affy\_read32* (*FILE \*input*, *void \*buf*) [File Access]  
Given an open input filestream *input* and a valid storage location pointed to by *buf*, four bytes will be read and stored in *buf*. The input stream is assumed to be in little-endian format; if the host platform is big-endian, the bytes will be swapped automatically.  
On success, '0' is returned. On failure, '-1' is returned and the contents of the target buffer are undefined. The caller is responsible for clearing or handling errors on the filestream.
- int** *affy\_read64* (*FILE \*input*, *void \*buf*) [File Access]  
Given an open input filestream *input* and a valid storage location pointed to by *buf*, eight bytes will be read and stored in *buf*. The input stream is assumed to be in little-endian format; if the host platform is big-endian, the bytes will be swapped automatically.  
On success, '0' is returned. On failure, '-1' is returned and the contents of the target buffer are undefined. The caller is responsible for clearing or handling errors on the filestream.
- int** *affy\_readchars* (*FILE \*input*, *char \*buf*, *size\_t numbytes*) [File Access]  
Given an open input filestream *input*, and a valid storage location pointed to by *buf* and of size *numbytes*, at most *numbytes*-1 bytes will be read and stored in *buf*. A 'NUL' byte is then appended to the result.  
This is primarily a convenience function for reading character data into C-style strings; it is similar to *fgets()* except that newline characters have no special significance, only encountering an 'EOF' or I/O error will cause a short read.

On success, '0' is returned. On failure, '-1' is returned and the contents of the target buffer are undefined. The caller is responsible for clearing or handling errors on the filestream.

`int affy_readint(FILE *input)` [File Access]

Given an open input filestream `input`, read what Affymetrix defines as an int (a 32-bit integer) and return as an int type. This is an unsafe function that will silently return when errors occur.

`int affy_readushort(FILE *input)` [File Access]

Given an open input filestream `input`, read what Affymetrix defines as a short (a 16-bit integer) and return as an int type. This is an unsafe function that will silently return when errors occur.

`int affy_readshort(FILE *input)` [File Access]

Given an open input filestream `input`, read what Affymetrix defines as a short (a 16-bit integer) and return as an int type. This is an unsafe function that will silently return when errors occur.

`float affy_readfloat(FILE *input)` [File Access]

Given an open input filestream `input`, read what Affymetrix defines as a float (a 32-bit floating point number) and return as a float type. This is an unsafe function that will silently return when errors occur.

## 5.2 Statistical Functions

`void affy_mean_normalization (AFFY_CHIPSET *d, double target_mean)` [Statistical]

Normalize an `AFFY_CHIPSET` containing one or more chips such that the mean intensity the chip (both perfect-match and mis-match probes) is equal across chips. The parameter `target_mean` is used as the mean intensity for all chips. Note this is not a trimmed mean and is performed on raw PM/MM values not expressions.

`double affy_median_save (double *x, int length)` [Statistical]

Calculate the median of `x` without disturbing the original ordering of `x`. The `length` of `x` must be specified.

`double affy_median (double *x, int length)` [Statistical]

Calculate the median of `x` but does not preserve the ordering of `x`. This saves allocating additional storage but allows a re-ordering of the original list.

Side effect: The order of `x` will be changed.

`void affy_get_row_median (double **z, double *rdelta, int rows, int columns)` [Statistical]

Within the median polish step of RMA, calculate the median of each row of matrix `x`, return median values in `rdelta` which must be allocated by the caller. The dimensions of the matrix are provided as `rows` and `columns`.

`void affy_get_column_median (double **z, double *cdelta, int rows, [Statistical]  
int columns)`

Within the median polish step of RMA, calculate the median of each column in matrix `z`, returning the median values in `cdelta` which must be allocated by the caller. The dimensions of the matrix are provided as `rows` and `columns`.

`int affy_median_sort (const void, *p1 const, void *p2) [Statistical]`

Helper function for `qsort` that sorts based on pointers to double values, used for median calculations.

`int affy_qnorm_compare (const void, *p1 const, void *p2) [Statistical]`

Helper function for `qsort` that sorts based on pointers to pointers of double values, used for quantile normalization.

`void affy_rank_order (double *rank, double **x, int n) [Statistical]`

Gets ranks in a process identical to that of the R statistical package. It assumes that `x` of length `n` is sorted, and calculates the ranks taking into account ties. Results are stored in `rank`.

`void affy_quantile_normalization (AFFY_CHIPSET *d) [Statistical]`

This function differs from the RMA quantile normalization, in that the normalization is performed on all probes, not just the Perfect Match (PM) probes. Otherwise the process is identical.

`void affy_pnorm_both (double x, double *cum, double *ccum, int [Statistical]  
i_tail, int log_p)`

This is a function from the R statistics software, which is freely usable under the GNU license. It calculates the distribution function of the normal distribution. This is used as part of the background correction within RMA.

`void rma_set_defaults (RMA_FLAGS *f) [Statistical]`

This routine sets flag structure `f` values to the defaults determined by the maintainers of `libaffy`.

`RMA_FLAGS *rma_get_defaults () [Statistical]`

Returns a dynamically allocated `RMA_FLAGS` structure with defaults filled in.

`AFFY_CHIPSET *rma (char **filelist, RMA_FLAGS *f) [Statistical]`

`void rma_background_correct (AFFY_CHIPSET *c, int chipnum) [Statistical]`

This procedure subtracts an intensity-dependent background value from all probes, for chip `chipnum` in the chipset `c`. Consult the RMA publications for the precise details of this algorithm.

`void quantile_normalization_chip (AFFY_CHIPSET *c, int chipnum, [Statistical]  
double *mean)`

Perform a quantile normalization on chip `chipnum` of the chipset `c`. The entire process cannot be performed for a single chip, however the individual chips can be sorted and quantile means can be calculated and updated in the array `mean`. The probe values are not adjusted within this function, only `mean` values are accumulated.

`void quantile_normalization_chipset (AFFY_CHIPSET *c, double [Statistical]  
*mean)`

Perform the final step of a quantile normalization across an entire chipset. The values in `mean` represent the mean values for each quantile of the Perfect Match array. Perfect Match probe values are replaced by the values in `mean` for each chip, after sorting to match quantiles with means.

`void rma_signal (AFFY_CHIPSET *c, RMA_FLAGS *f) [Statistical]`

Calculate the signal for a probeset, from a set of probes, for all chips in `c`. This is done by defining a model of probes for a probeset, and fitting the data to this model. The model-fitting is performed via a median polish. Flags that modify the default behavior of this algorithm can be passed via `f`.

`void median_polish (double **z, int numprobes, int numchips, double [Statistical]  
*results, double *affinities)`

Uses the median polish algorithm to fit a model consisting of co-efficients given in `z`, a `numprobes` by `numchips` matrix. The results are stored for return in `results` and the residuals are return in `affinities`. The `results` are the actual probeset signal and the `affinities` are representative of the estimated binding efficiency of each probe.

`MAS5_FLAGS *mas5_get_defaults () [Statistical]`

Returns a dynamically allocated `MAS5_FLAGS` structure with defaults filled in.

`void mas5_set_defaults (MAS5_FLAGS *f) [Statistical]`

This routine sets flag structure `f` values to the defaults determined by the maintainers of `libaffy`.

`AFFY_CHIPSET *mas5 (char **filelist, MAS5_FLAGS *f) [Statistical]`

Performs the MAS 5.0 algorithm for calculating expression over a list of files given by `filelist`. Flags modifying the default behavior of MAS 5.0 can be given as `f` (or `NULL` for defaults).

`int mas5_background_correction (AFFY_CHIPSET *c, MAS5_FLAGS [Statistical]  
*f)`

Performs the Affymetrix MAS 5.0 background correction on the chips in `c`. Flags that may modify the algorithm are provided in `f`. This process involves calculating zones and subtracting weighted averages from different zones. Consult the Affymetrix documentation for the details of this algorithm.

`int mas5_signal (AFFY_CHIPSET *c, MAS5_FLAGS *f) [Statistical]`

Use the MAS 5.0 algorithm to calculate signal values, for the chipset `c`. Flags that may modify the algorithm are given in `f`. This process calculates Tukey's biweight on the difference from perfect match and ideal mismatch.

`int mas5_scale (AFFY_CHIPSET *c, MAS5_FLAGS *f) [Statistical]`

Performs the Affymetrix scaling function, which normalizes expression values such that the trimmed mean of each chip in `c` is set to a target value. This value is defined as `f->scale_target` with upper and lower bounds defined by `f->trimmed_mean_low` and `f->trimmed_mean_high`.

### 5.3 Utility Functions

- char** *\*affy\_get\_cdf\_name*(*char \*buf*) [Utility]  
 Given the ‘DatHeader’ portion of a CEL file in *buf*, the corresponding CDF filename is returned.  
 New storage is allocated for the resultant string, which must be freed by the caller. NULL is returned in the case of insufficient memory.
- char** *\*affy\_get\_cdf\_name\_from\_cel*(*char \*filename*) [Utility]  
 Given a CEL filename *filename*, open the CEL file, determine the corresponding CDF filename, and return the resultant string.  
 The library will end execution with a fatal exception on I/O error.  
 New storage is allocated on success, which must be freed by the caller. NULL is returned in the case of insufficient memory.
- double** *\*\*affy\_matrix\_from\_cel*(*AFFY\_CELFILE \*cf*) [Utility]  
 This function is provided as a convenience to quickly extract the cell intensity values from a CEL file without dealing with individual cell structures. Given an initialized CEL file structure *cf*, the cell intensity values are copied into a two-dimensional array and the resultant pointer is returned. The array is guaranteed to be contiguous in memory.  
 If the array could not be allocated due to insufficient memory, NULL is returned.
- void** *affy\_dump\_dat\_hdr*(*AFFY\_DATFILE \*df*) [Utility]  
 Given an initialized DAT file structure *df*, a human-readable summary of the file header information is written to standard output. Fields are written one per line. It should be noted that not all fields necessarily contain valid data in every DAT file, but they will be printed regardless.
- AFFY\_PIXELREGION** *\*affy\_pixels\_from\_cell*(*AFFY\_CHIP \*cp, int x, int y*) [Utility]  
 Given a *AFFY\_CHIP* structure *cp* containing at least valid CDF and DAT structures, return the pixels associated with cell coordinate (*x,y*). Rotation of the grid is considered, as defined by corner coordinates within the DAT file.  
 If the coordinates are not valid, a NULL pointer is returned.
- AFFY\_POINT** *affy\_cell\_to\_pixel*(*AFFY\_CHIP \*cp, int x, int y*) [Utility]  
 Given a *AFFY\_CHIP* structure *cp* containing at least valid CDF and DAT structures, return the image coordinates corresponding to upper left corner of the cell. This takes into account rotation of the grid.
- void** *affy\_write\_expressions*(*AFFY\_CHIPSET \*c, char \*filename*) [Utility]  
 Opens the platform-dependent filename given by *filename* and writes out the gene expression data from each chip in the chipset *c*.  
 The library will halt execution with a fatal exception on error.
- void** *affy\_write\_pixel\_region*(*AFFY\_PIXELREGION \*pr, char \*filename*) [Utility]  
 Writes the *AFFY\_PIXELREGION* *pr* to a platform-dependent file given by *filename*. If TIFF image support is compiled into *libaffy*, the output will be a tiff image, otherwise a tab-delimited text file containing the intensities in the region.

**Boolean** *affy\_ismasked* (*AFFY\_CHIP \*chip, int x, int y*) [Utility]

Given an initialized chip structure *chip* and cell coordinates *x* and *y*, test whether the cell is masked.

**True** is returned if the cell is masked, **False** if not.

**Boolean** *affy\_iscontrol* (*AFFY\_CHIP \*chip, int x, int y*) [Utility]

Given an initialized chip structure *chip* and cell coordinates *x* and *y*, test whether the cell is a control location.

**True** is returned if the cell is a control location, **False** if the cell is a normal location.

# Concept Index

(Index is nonexistent)