CALIB Manual

for version 1.0.2, 21 July, 2025

David M. Warme

This manual is for CALIB (version 1.0.2, 21 July, 2025), a software library for computer algebra.

Table of Contents

1 Introduction to CALIB.

CALIB stands for Computer Algebra LIBrary. It is a software library that can be invoked by and linked with other programs, providing various algorithms performing algebraic operations. Unlike typical programming languages or calculators (that perform arithmetic only on "numbers"), CALIB's operations are *symbolic* — like one would work algebra or calculus problems using pencil and paper.

CALIB is an experiment in one particular manner of *structuring* a collection of algorithms for computer algebra. Having examined a few other computer algebra systems, the author has found some of these to be "architectural messes" internally. CALIB is an attempt to bring some order to this perceived chaos. Your mileage may vary regarding the extent to which CALIB has succeeded with this goal.

CALIB does not claim to provide algorithms that are state-of-the-art. Some of the methods are ancient, while others are fairly modern. CALIB includes sample applications that demonstrate its practical utility at solving real-world algebraic problems with quite reasonable efficiency.

2 Building and Installing CALIB.

How to build and install CALIB.

2.1 Basic Installation

CALIB comes with a “GNU style” configure script. For those of you who are especially impatient, type the following:

```
./configure
make
```

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create an `Minit.mk` file used by each `Makefile` of the package. It also creates a `config.h` file containing system-dependent definitions. Finally, it creates a shell script `config.status` that you can run in the future to recreate the current configuration, a file `config.cache` that saves the results of its tests to speed up reconfiguring, and a file `config.log` containing compiler output (useful mainly for debugging `configure`).

If you need to do unusual things to compile CALIB, please try to figure out how `configure` could check whether and how to do them, and mail diffs or instructions to the address given in the `README` so they can be considered for our next release.

The file `configure.in` is used to create `configure` by a program called `autoconf`. You only need `configure.in` if you want to change it or regenerate `configure` using a newer version of GNU `autoconf`.

NOTE: you do NOT need the GNU `autoconf` program unless you plan to change the `configure.in` file!!!

The simplest way to compile this package is:

1. `cd` to the "top-level" directory containing CALIB's source code and other distributed files. (This directory contains this `INSTALL` file, `LICENSE`, `README` and various other files and sub-directories.) Then type `./configure` to configure CALIB for your system. If you're using `csh` on an old version of System V, you might need to type `sh ./configure` instead to prevent `csh` from trying to execute `configure` itself.

   Running `configure` prints various messages telling which features it is checking for, and various options it has chosen.

2. Type `make` to compile CALIB.
3. The CALIB library, header files and sample applications will execute properly right in the build directory. However, if you want to install CALIB in a more permanent place (`/usr/local`, or whichever `--prefix` option you gave to `configure`), then type `make install` to install the programs and data files. Of course, you need to have write permission on these directories or this will not work.
4. You can remove the program binaries and object files from the source code directory by typing `make clean`. To also remove the files that `configure` created (so you can compile the package for a different kind of computer), type `make distclean`.

2.2 Compilers and Options

Hopefully you have the GNU C compiler (`gcc`). This is what we use, and CALIB compiles cleanly with gcc.

Some systems require unusual options for compilation or linking that the `configure` script does not know about. You can give `configure` initial values for variables by setting them in the environment. Using a Bourne-compatible shell, you can do that on the command line like this:

```
CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure
```

Or on systems that have the `env` program, you can do it like this:

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure
```

2.3 GNU Multi-Precision Arithmetic Library (GMP)

CALIB depends upon GMP (the GNU Multi-Precision arithmetic package) to provide arbitrary precision arithmetic for integer and rational numbers. GMP can be downloaded from:

```
http://www.gnu.org/software/
```

It must be installed prior to building CALIB. (Many Linux distributions provide packages for GMP. On RPM-based systems such as Red Hat Enterprise Linux and Fedora, the `gmp` and `gmp-devel` RPMs provide what CALIB needs.)

CALIB assumes that GMP has been installed in the "system-wide" locations so that

```
#include <gmp.h>
```

works, and one can simply use `-lgmp -lm` to link GMP into an application program. It is therefore best to use the version of GMP provided by your Linux distro, if possible.

2.4 The Van Hoeij Algorithm, LLL and FPLLL

CALIB provides an implementation of the Van Hoeij algorithm that factors polynomials in Z[x] (single-variable polynomials with integer coefficients) in polynomial time. The Van Hoeij algorithm in turn requires an implementation of LLL: the Lenstra-Lenstra-Lovász lattice basis reduction algorithm. If a suitable LLL implementation is available, CALIB will automatically provide the polynomial time Van Hoeij algorithm for factoring over Z[x]. (Otherwise it falls back to the older Zassenhaus algorithm that can take exponential time on certain classes of polynomials.)

The LLL algorithm is very challenging to implement in a manner that provides competitive computational performance. (CALIB's own implementation of LLL is not yet ready for prime time.) CALIB therefore uses `FPLLL` — a high-performance, floating-point implementation of LLL that is highly tuned. If you want CALIB to use the Van Hoeij algorithm, then you will have to download and build a sufficiently recent version of FPLLL from here:

```
https://github.com/fplll/fplll
```

Build FPLLL according to the instructions provided. Then do the following to configure CALIB to use FPLLL:

```
./configure --with-fplllheaderdir=H --with-fplllLib=L
```

where

`H` is the *directory* containing FPLLL's header files (you have the correct directory if the file `H/fplll/wrapper.h` exists); and

`L` is the the path name of the FPLLL library file (`libfplll.a`).

(It is probably best to specify `H` and `L` as absolute path names.) And of course you should also provide any additional arguments to the `configure` script, such as `--prefix=PATH`. Once again, use of FPLLL is entirely optional.

2.5 Installation Names

By default, `make install` will install the package's files in `/usr/local/bin`, `/usr/local/man`, etc. You can specify an installation prefix other than `/usr/local` by giving `configure` the option `--prefix=PATH`.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you give `configure` the option `--exec-prefix=PATH`, the package will use `PATH` as the prefix for installing programs and libraries. Documentation and other data files will still use the regular prefix.

In addition, if you use an unusual directory layout you can give options like `--bindir=PATH` to specify different values for particular kinds of files. Run `configure --help` for a list of the directories you can set and what kinds of files go in them.

If the package supports it, you can cause programs to be installed with an extra prefix or suffix on their names by giving `configure` the option `--program-prefix=PREFIX` or `--program-suffix=SUFFIX`.

2.6 Sharing Defaults

If you want to set default values for `configure` scripts to share, you can create a site shell script called `config.site` that gives default values for variables like `CC`, `cache_file`, and `prefix`. `configure` looks for `PREFIX/share/config.site` if it exists, then `PREFIX/etc/config.site` if it exists. Or, you can set the `CONFIG_SITE` environment variable to the location of the site script. A warning: not all `configure` scripts look for a site script.

2.7 Operation Controls

`configure` recognizes the following options to control how it operates.

`--cache-file=FILE`
: Use and save the results of the tests in `FILE` instead of `./config.cache`. Set `FILE` to `/dev/null` to disable caching, for debugging `configure`.

`--help`
: Print a summary of the options to `configure`, and exit.

`--quiet`
`--silent`
`-q`
: Do not print messages saying which checks are being made.

`--srcdir=DIR`
: Look for the package's source code in directory `DIR`. Usually `configure` can determine that directory automatically.

`--version`
: Print the version of Autoconf used to generate the `configure` script, and exit.

`configure` also accepts some other, not widely useful, options.

3 Basic Conventions for Using CALIB.

CALIB defines a set of *algebraic domains.* Each domain `D` provides a set of *operations* that can be performed upon algebraic *values* (i.e., objects) that are *members* of domain `D` (or other domains).

In most cases, a domain `D` must be specified by *instantiating* it with certain parameters. For example, when creating a Zp domain (the integers modulo a prime p), one must specify a particular prime p. For example:

```
struct calib_Zp_dom * dom;

dom = calib_make_Zp_dom_ui (23);
```

creates a domain `dom` representing "the integers modulo 23." All future operations invoked via `dom` understand that the prime modulus being used is 23. If one invokes `dom -> add (...)` or `dom -> inv (...)`, it will be understood to mean "addition modulo 23" or "multiplicative inverse modulo 23," respectively.

Most of CALIB's operations exist as "member functions" (i.e., members that are pointers to functions) residing in the corresponding domain object.

The fundmental architectural idea being explored in CALIB is this notion of *operations* provided by *domain* objects that represent *completely specified* algebraic domains.

A more complicated CALIB domain can be constructed to represent the Galois Field $GF(p^k)$ defined by the polynomial $x^3 + 2x + 7$ which is a monic, irreducible polynomial in $Zp[x]$, with $p = 23$.

The central idea of CALIB is that all details of this algebraic structure (i.e., the prime $p = 23$ and the Galois Field generator polynomial) are fully encapsulated within a hierarchically structured set of domain objects that contain all of the instantiated data (e.g., prime `p = 23` and generator polynomial $x^3 + 2x + 7$). The constructed domain object then provides operations/algorithms operating over the "values" of that domain.

Domain objects in calib all use the following naming convention:

```
struct calib_FOO_dom;
```

where `FOO` denotes the type of algebraic domain.

The algebraic *values* that represent members or instances of a domain are usually represented by a second object, conventionally named as follows:

```
struct calib_FOO_obj;
```

Exceptions to this rule occur for domains whose values are represented directly as GMP integers or rationals. (The `Zp` domain is the only current exception to this rule. `Zp` does not provide a `struct calib_Zp_obj` "value object," but instead represents its values directly as GMP integers.)

CALIB has adopted the GMP conventions for value object construction and destruction: providing `init()` and `clear()` operations to initialize raw value objects in the domain. In many cases, the ability to initialize and clear given arrays of such objects are also provided. These conventions permit the value objects (if desired) to be local variables of a function, efficiently stored in the stack frame of the function, rather than dynamically allocated from the memory heap.

Unless otherwise specified, the “value” of a value object after initialization is zero.

CALIB value objects always contain the domain to which they belong, which must be specified at value object `init()` time. This has at least two direct implications:

- There is no need to pass the domain as an argument to most operations, since each value object already contains the domain to which it belongs; and
- Operations taking two or more such value objects must check the domains of its various operands to verify these domains are compatible. For most operations, the domains of all operands must be identical (same pointer address).

Many of CALIB’s domains provide a `dup()` function that returns a dynamically-allocated copy of the given source value, and a corresponding `free()` function that combines the `clear()` operation with a call to the standard C library `free()` function.

There is no garbage collection with CALIB. It uses standard C programming methods to manage memory usage with all of the corresponding advantages (efficiency) and pitfalls (memory leaks). Good tools are available for detecting memory leaks, and they seem to work well with CALIB.

3.1 CALIB Types

The header file `"calib/types.h"` provides the following typedefs:

```
typedef signed char calib_int8s;
typedef unsigned char calib_int8u;
typedef char calib_int8;

typedef signed short calib_int16s;
typedef unsigned short calib_int16u;
typedef short calib_int16;

typedef signed int calib_int32s;
typedef unsigned int calib_int32u;
typedef int calib_int32;

typedef signed long long calib_int64s;
typedef unsigned long long calib_int64u;
typedef long long calib_int64;

typedef char calib_bool;

/* Types mimicking those that "should" be in GMP (but are not). */

typedef signed long int calib_si_t;
typedef unsigned long int calib_ui_t;
```

These provide 8, 16, 32 and 64-bit integers, with versions that are specifically `unsigned`, `signed` and the “natural” signedness of the given type. (For example, the C standard gives implementations freedom for `char` to be either `signed` or `unsigned`.) The designers of GMP missed an opportunity to define `typedef`s like `gmp_ui_t` and `gmp_si_t` to encapsulate the types of "raw," language-level `unsigned` and `signed` integer types used in the GMP function type signatures. CALIB avoids this pitfall by providing `calib_ui_t` and `calib_si_t`.

3.2 Argument Passing Conventions

Unless otherwise specified, the same address can be passed to arguments that are pointers to objects of the same type. The following is an example where two inputs and an output argument all refer to the same object `POLY`:

```
const struct calib_Zx_dom * Zx;
struct calib_Zx_obj POLY;
Zx = calib_get_Zx_dom ();
Zx -> init (&POLY);
...
/* This squares the Z[x] polynomial POLY. */
Zx -> mul (&POLY, &POLY, &POLY);
```

For functions having multiple output arguments, however, it is *never* permitted to pass the same (non-`NULL`) address to two or more such output arguments — the result in such cases would depend upon the order in which the CALIB function assigns these output values. CALIB chooses to leave such assignment order explicitly undefined.

4 Z and Q — The Integer and Rational Numbers.

CALIB uses `GMP` (The GNU Multi-Precision arithmetic package) to provide all underlying arithmetic operations for both integer and rational numbers of arbitrary precision. To use GMP, one must first

```
#include <gmp.h>
```

For arbitrary precision integers, GMP provides types `mpz_t`, `mpz_ptr` and `mpz_srcptr`. All of the associated functions and macros begin with the `mpz_` prefix.

For arbitrary precision rationals, GMP provides types `mpq_t`, `mpq_ptr` and `mpq_srcptr`. All of the associated functions and macros begin with the `mpq_` prefix.

Refer to the GMP documentation for full details.

5 CALIB General Representation

CALIB provides a *general representation* (or “genrep” for short) that can hold any type of symbolic expression handled by CALIB. One may access CALIB’s genrep facilities as follows:

```
#include "calib/genrep.h"
```

For example, it is possible to read a symbolic expression from a text file yielding the corresponding genrep `gp`, which can then be converted into a value in various other CALIB algebraic domains. Similarly, each CALIB algebraic domain `D` provides the ability to convert a given value of `D` into genrep form.

CALIB provides various functions to “pretty print” expressions in genrep form, and print them to files in various syntaxes.

The general representation is therefore the “common format” by which the various CALIB algebraic domains communicate with the outside world. Other than the various expression parsers, printers, pretty-printers and functions to construct/destruct various flavors of primitive genrep objects, no substantive “algebraic algorithms” (beyond a few simplifications) are provided for expressions in the genrep form. The general representation is primarily a communication medium between the various parts of CALIB.

The general representation consists of several types of objects, as follows:

```
struct calib_genrep {
char op; /* Operator. */
union {
struct calib_genrep_list * list; /* List of operand subexprs */
char * var; /* Variable */
calib_si_t siop; /* Signed integer */
mpz_ptr zop; /* Integer operand */
mpq_ptr qop; /* Rational operand */
struct {
struct calib_genrep * base;
int power;
} ipow;
struct {
struct calib_genrep * func;
struct calib_genrep_list * args;
} func;
} u;
};

struct calib_genrep_list {
struct calib_genrep * operand; /* Current operand */
struct calib_genrep_list * next; /* List of other operands */
};
```

The genrep `op` field has one of the following symbolic values (`#define`’d in `calib/genrep.h`):

```
CALIB_GENREP_OP_ADD /* sum of zero or more genreps */
```

```
CALIB_GENREP_OP_MUL /* product of zero or more genreps */
CALIB_GENREP_OP_IPOW /* integer power of a genrep */
CALIB_GENREP_OP_VAR /* a single variable */
CALIB_GENREP_OP_SI /* a signed integer */
CALIB_GENREP_OP_Z /* a GMP integer */
CALIB_GENREP_OP_Q /* a GMP rational */
CALIB_GENREP_OP_FUNC /* a function invocation */
```

The `u.list` member is used for sums and products, which each permit a linked-list of zero or more sub-operand genreps.

One additional structure is provided in the genrep header:

```
struct calib_genrep_varlist {
int nvars; /* Number of vars in the list */
const char * const * vlist; /* List of variable names.  Note */
/* that vlist[nvars] EQ NULL. */
};
```

This object is used to hold a (usually alphabetized) list of all the variables appearing within a given genrep.

The `"calib/genrep.h"` header provides the following global functions:

calib_genrep_add2:

```
struct calib_genrep *
calib_genrep_add2 (const struct calib_genrep * op1,
   const struct calib_genrep * op2);
```

Returns a dynamically allocated genrep for the sum of the two given genreps, where:

`op1` is the first genrep operand; and
`op2` is the second genrep operand.

calib_genrep_clear_varlist:

```
void calib_genrep_clear_varlist (
struct calib_genrep_varlist * list);
```

Clear out the given genrep varlist, where:

`list` is the genrep varlist to be cleared.

calib_genrep_convert_abs_Z_to_decimal_string:

```
size_t calib_genrep_convert_abs_Z_to_decimal_string (
mpz_srcptr zval,
char ** str_out);
```

Sets `*str_out` to a nul-terminated, dynamically allocated string containing the absolute value of `zval` as decimal digits, and returning the number of decimal digits in the string, where:

`zval` is the integer whose absolute value is to be converted; and
`str_out` is the address of the `char *` variable to receive the dynamically-allocated string.

It is the responsibility of the caller to `free()` the string it receives from this function.

calib_genrep_div2:

```
struct calib_genrep *
calib_genrep_div2 (const struct calib_genrep * op1,
   const struct calib_genrep * op2);
```

Returns a dynamically allocated genrep for `op1 / op2`, where:

`op1` is the first genrep operand; and
`op2` is the second genrep operand.

This function will gladly produce a genrep that (symbolically) represents division by zero.

calib_genrep_dup:

```
struct calib_genrep *
calib_genrep_dup (const struct calib_genrep * op);
```

Returns a dynamically-allocated “deep copy” of the given genrep, where:

`op` is genrep to be recursively duplicated.

calib_genrep_dup_list:

```
struct calib_genrep_list *
calib_genrep_dup_list (const struct calib_genrep_list * list);
```

Returns a dynamically-allocated “deep copy” of the given genrep_list, where:

`list` is genrep_list to be recursively duplicated.

calib_genrep_fread:

```
struct calib_genrep *
calib_genrep_fread (FILE * fp, int * status);
```

Read an expression (terminated by a semicolon) from the given input stream, returning it as a dynamically-allocated genrep, where:

`fp` is the input stream from which to read the expression; and
`status` is the address of an `int` variable set to zero upon success and one if a syntax error is encountered.

Valid combinations of (return value, status) are as follows:

(NULL, 0)	End of file.
(NULL, 1)	Syntax error in expression.
(non-NULL, 0)	Expression read successfully.

Note: `calib_genrep_fread` supports both C and C++ style comments. Unless the input stream is a (presumably interactive) TTY, such comments are automatically echoed to stdout.

calib_genrep_free:

```
void calib_genrep_free (struct calib_genrep * op);
```

Recursively free the given genrep, where:

`op` is genrep to be recursively freed.

calib_genrep_free_list:

```
void calib_genrep_free_list (struct calib_genrep_list * list);
```

Recursively free the given genrep_list, where:

`list` is genrep_list to be recursively freed.

calib_genrep_fprint:

```
void calib_genrep_fprint (FILE * fp,
     const struct calib_genrep * op);
```

Display the given genrep to the given output stream using the default maximum width, where:

`fp` is the output stream to write into; and
`op` is genrep to be written.

Note: this is an old "pretty printer" that is obsolete because it considers many expressions to be "too wide to print."

calib_genrep_fwprint:

```
void calib_genrep_fwprint (FILE * fp,
      const struct calib_genrep * op,
      int width);
```

Display the given genrep to the given output stream using the given maximum width, where:

`fp` is the output stream to write into;
`op` is genrep to be written; and
`width` is the maximum width to use.

Note: this is an old "pretty printer" that is obsolete because it considers many expressions to be "too wide to print."

calib_genrep_get_varlist:

```
void calib_genrep_get_varlist (
struct calib_genrep_varlist * list,
const struct calib_genrep * op);
```

Scan the given genrep to determine the names of all variables appearing within the expression, then fill in the given varlist with a sorted list of the variable names, where:

`list` is the genrep varlist to be filled in; and
`op` is the genrep to be recursively scanned for variable names.

calib_genrep_ipow:

```
struct calib_genrep *
calib_genrep_ipow (const calib_genrep * op1, int op2);
```

Returns a dynamically-allocated genrep `opt1^op2`, where:

`op1` is the genrep to be exponentiated; and
`op2` is the integer power to use.

calib_genrep_mul2:

```
struct calib_genrep *
calib_genrep_mul2 (const struct calib_genrep * op1,
   const struct calib_genrep * op2);
```

Returns a dynamically allocated genrep for the product of the two given genreps, where:

`op1` is the first genrep operand; and
`op2` is the second genrep operand.

calib_genrep_neg:

```
struct calib_genrep *
calib_genrep_neg (const struct calib_genrep * op);
```

Returns a dynamically allocated genrep for the negative of the given genrep, where:

`op` is the genrep operand to be negated.

calib_genrep_new_list:

```
struct calib_genrep_list *
calib_genrep_new_list (struct calib_genrep * op,
       struct calib_genrep_list * next);
```

Return a dynamically-allocated genreplist whose `operand` and `next` fields are given, where:

`op` is the new genrep_list's `operand` field; and
`next` is the new genrep_list's `next` field.

calib_genrep_poly_term:

```
struct calib_genrep *
calib_genrep_poly_term (struct calib_genrep * coeff,
const char * var,
int pow);
```

Return a dynamically-allocated genrep that represents `coeff*var^pow`, where:

`coeff` is the coefficient of the polynomial term;
`var` is the polynomial variable; and
`pow` is the exponent (degree) of the polynomial term.

calib_genrep_prettyprint:

```
void calib_genrep_prettyprint (const struct calib_genrep * op);
```

Pretty-print the given genrep to stdout using the default width (determined from the terminal width if stdout refers to some sort of tty whose width can be determined), where:

`op` is the genrep to be prettyprinted.

This uses the new prettyprinting algorithm that uses dynamic programming to optimize the layout of the various sub-expression tiles.

calib_genrep_prettyprint_file:

```
void calib_genrep_prettyprint_file (
FILE * fp,
const struct calib_genrep * op);
```

Pretty-print the given genrep to the given output stream using the default width (determined from the terminal width if the given output stream refers to some sort of tty whose width can be determined), where:

`fp` is the output stream into which the expression is prettyprinted; and
`op` is the genrep to be prettyprinted.

This uses the new prettyprinting algorithm that uses dynamic programming to optimize the layout of the various sub-expression tiles.

calib_genrep_prettyprint_file_width:

```
void calib_genrep_prettyprint_file_width (
FILE * fp,
const struct calib_genrep * op,
int width);
```

Pretty-print the given genrep to the given output stream using the given width, where:

`fp` is the output stream into which the expression is prettyprinted;
`op` is the genrep to be prettyprinted; and
`width` is the maximum display width to use.

This uses the new prettyprinting algorithm that uses dynamic programming to optimize the layout of the various sub-expression tiles.

calib_genrep_print_maxima:

```
void calib_genrep_print_maxima (
FILE * fp,
const struct calib_genrep * op,
int width);
```

Print the given genrep (in syntax readable by Maxima) to the given output stream using the given maximum line width, where:

`fp` is the output stream into which the expression is printed;
`op` is the genrep to be printed; and
`width` is the maximum line width to use.

calib_genrep_prettyprint_width:

```
void calib_genrep_prettyprint_width (
const struct calib_genrep * op,
int width);
```

Pretty-print the given genrep to stdout using the given width, where:

`op` is the genrep to be prettyprinted; and
`width` is the maximum display width to use.

This uses the new prettyprinting algorithm that uses dynamic programming to optimize the layout of the various sub-expression tiles.

calib_genrep_print:

```
void calib_genrep_print (const struct calib_genrep * op);
```

Display the given genrep to stdout using the default maximum width, where:

`op` is genrep to be written.

Note: this is an old "pretty printer" that is obsolete because it considers many expressions to be "too wide to print."

calib_genrep_q:

```
struct calib_genrep *
calib_genrep_ (mpq_srcptr op);
```

Returns a dynamically-allocated genrep representing the given rational value, where:

`op` is the rational value to be converted into genrep form.

calib_genrep_read:

```
struct calib_genrep *
calib_genrep_read (int * status);
```

Read an expression (terminated by a semicolon) from stdin, returning it as a dynamically-allocated genrep, where:

`status` is the address of an `int` variable set to zero upon success and one if a syntax error is encountered.

Valid combinations of (return value, status) are as follows:

(NULL, 0)	End of file.
(NULL, 1)	Syntax error in expression.
(non-NULL, 0)	Expression read successfully.

Note: `calib_genrep_read` upports both C and C++ style comments. Unless the input stream is a (presumably interactive) TTY, such comments are automatically echoed to stdout.

calib_genrep_si:

```
struct calib_genrep *
calib_genrep_si (calib_si_t op);
```

Returns a dynamically-allocated genrep representing the given integer value, where:

`op` is the integer value to be converted into genrep form.

calib_genrep_sub2:

```
struct calib_genrep *
calib_genrep_sub2 (const struct calib_genrep * op1,
   const struct calib_genrep * op2);
```

Returns a dynamically allocated genrep for the difference of the two given genreps, where:

`op1` is the first genrep operand; and
`op2` is the second genrep operand.

calib_genrep_var:

```
struct calib_genrep *
calib_genrep_var (const char * var);
```

Returns a dynamically-allocated genrep representing the given variable, where:

`var` is the variable to be converted into genrep form.

calib_genrep_wprint:

```
void calib_genrep_wprint (const struct calib_genrep * op,
     int width);
```

Display the given genrep to stdout using the given maximum width, where:

`op` is genrep to be written; and
`width` is the maximum width to use.

Note: this is an old “pretty printer” that is obsolete because it considers many expressions to be “too wide to print.”

calib_genrep_z:

```
struct calib_genrep *
calib_genrep_z (int op);
```

Returns a dynamically-allocated genrep representing the given GMP integer value, where:

`op` is the GMP integer value to be converted into genrep form.

5.1 Genrep Functions

The `u.func.func` member of `struct calib_genrep` must be a genrep having `op` that is one of the following:

- `CALIB_GENREP_OP_VAR` represents a general function name as a variable / string.
- `CALIB_GENREP_OP_SI` containing an index representing one of the CALIB “builtin” functions.

The CALIB builtin functions are as follows:

```
exp log abs sqrt
sin cos tan cot sec csc
asin acos atan acot asec acsc
sinh cosh tanh coth sech csch
asinh acosh atanh acoth asech acsch
atan2 integrate
```

For each function `foo` in this list, calib provides a corresponding `#define` for the function index of the form `CALIB_GENREP_FUNC_FOO`, and a corresponding function `calib_genrep_foo(x)` that returns a genrep representing `foo(x)`. (This function takes one, two or more arguments as appropriate for the particular function. For example `calib_genrep_atan2 (y, x)` takes two args.)

CALIB does not currently guarantee these builtin function indices to have permanent and stable values with respect to the CALIB ABI. It is therefore recommended that the functions be used; using the `#define` symbols may require recompilation with new CALIB releases.

CALIB provides two functions for translating between builtin function names and their indices:

calib_genrep_builtin_func_name_to_index:

```
calib_si_t
calib_genrep_builtin_func_name_to_index (const char * fname);
```

Return the "builtin function index" corresponding to the given `fname`, or -1 if no such builtin function exists.

calib_genrep_builtin_func_index_to_name:

```
const char *
calib_genrep_builtin_func_index_to_name (calib_si_t fidx);
```

Return the name of the builtin function having "builtin function index" `fidx`, or `NULL` if the given `fidx` is invalid.

CALIB provides the following additional functions for manipulating `CALIB_GENREP_OP_FUNC` genrep nodes:

calib_genrep_func:

```
struct calib_genrep *
calib_genrep_func (struct calib_genrep * f,
   struct calib_genrep_list * args);
```

Return a dynamically-allocated `CALIB_GENREP_OP_FUNC` node having function `f` and arguments `args`.

calib_genrep_func_si_1:

```
struct calib_genrep *
calib_genrep_func_si_1 (calib_si_t fidx,
struct calib_genrep * op);
```

Return a dynamically-allocated `CALIB_GENREP_OP_FUNC` node having builtin function index `fidx` and argument `op`.

calib_genrep_func_si_2:

```
struct calib_genrep *
calib_genrep_func_si_2 (calib_si_t fidx,
struct calib_genrep * arg1,
struct calib_genrep * arg2);
```

Return a dynamically-allocated `CALIB_GENREP_OP_FUNC` node having builtin function index `fidx` and two argument: `arg1` and `arg2`.

calib_genrep_func_str:

```
struct calib_genrep *
calib_genrep_func_str (const char * fname,
        struct calib_genrep_list * args);
```

Return a new dynamically-allocated `CALIB_GENREP_OP_FUNC` node having function named `fname` and arguments `args`.

6 Zx — The Polynomial Ring $Z[x]$

CALIB provides the `Zx` domain, representing the ring $Z[x]$, the univariate polynomials having integer coefficients. The "values" of this domain are represented by the following object:

```
struct calib_Zx_obj {
int degree; /* Degree of polynomial  */
int size; /* Size of coeff[] array (degree < size) */
mpz_ptr coeff; /* Coefficients of polynomial.   */
};
```

These objects are subject to `init()` and `clear()` operations. All such objects must be initialized prior to use by any other CALIB operation. Memory leaks result if they are not cleared when done.

The following additional object is used to represent linked-lists of factors produced by various Zx factorization algorithms:

```
struct calib_Zx_factor {
int multiplicity;
struct calib_Zx_obj * factor;
struct calib_Zx_factor * next;
};
```

No data is needed to instantiate the `Zx` domain. (There is only one such object, and cannot be freed.) Nonetheless, the `Zx` domain object is the only way to access the operations provided by this domain.

One may access CALIB's `Zx` domain as follows:

```
#include "calib/Zx.h"
...
        const struct calib_Zx_dom * Zx;
struct calib_Zx_obj poly1, poly2;
...
Zx = calib_get_Zx_dom ();
...
Zx -> init (&poly1);
Zx -> init (&poly2);
...
Zx -> mul (&poly1, &poly1, &poly2);
...
Zx -> clear (&poly2);
Zx -> clear (&poly1);
```

The CALIB `Zx` domain supports the following settings:

```
/*
 * Which factorization algorithm to use (for square-free polynomials).
 */

enum calib_Zx_factor_method {
```

```
CALIB_ZX_FACTOR_METHOD_ZASSENHAUS,
CALIB_ZX_FACTOR_METHOD_VAN_HOEIJ,
};

/*
 * Which method to use for lifting modular factors.
 */

enum calib_Zx_lift_method {
CALIB_ZX_LIFT_METHOD_LINEAR,
CALIB_ZX_LIFT_METHOD_QUADRATIC,
};

/*
 * Which method to use for gcd (gcd, a, b).  (The gcd_n() function always
 * uses modular.)
 */

enum calib_Zx_gcd_method {
CALIB_ZX_GCD_METHOD_MODULAR,
CALIB_ZX_GCD_METHOD_PRS
};

/*
 * The "settings" object for the Zx domain.
 */

struct calib_Zx_settings {
/* Factorization method for square-free Zx polynomials. */
enum calib_Zx_factor_method factor_method;

/* Method for lifting modular factors. */
enum calib_Zx_lift_method lift_method;

/* Method to use for two-operand gcd(). */
enum calib_Zx_gcd_method gcd_method;

/* If number of remaining modular factors does not exceed this, */
/* then use exhaustive enumeration instead of Van Hoeij.  */
int VH_max_enumerate;

/* Enable trace output from Z[x] factorization algorithm.  */
/* Zero is none.  Higher values give more output.  */
int factor_trace_level;

/* Van Hoeij uses resultant-based trace root bound? */
calib_bool VH_use_resultant_trace_root_bound;
```

```
/* Validate lifted modular factors? */
calib_bool validate_lifted_modular_factors;
};

/*
 * All settings for Zx domain are global.
 */

extern struct calib_Zx_settings calib_Zx_settings;
```

The `struct calib_Zx_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Zx::init():

```
void (*init) (struct calib_Zx_obj * x);
```

Initialize the given Zx polynomial `x`, where:

`x` is the polynomial to initialize.

Zx::init_si():

```
void (*init_si) (struct calib_Zx_obj * x,
    calib_si_t op);
```

Initialize the given Zx polynomial `x` to the given constant value `op`, where:

`x` is the polynomial to initialize; and
`op` is the constant value to which the polynomial is set.

Zx::alloc():

```
void (*alloc) (struct calib_Zx_obj * rop,
  int degree);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial of at least the given `degree`, where:

`rop` is the polynomial whose allocation is to be adjusted; and
`degree` is the guaranteed minimum degree polynomial that `rop` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Zx::clear():

```
void (*clear) (struct calib_Zx_obj * x);
```

Clear out the given polynomial `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x` is the polynomial to be cleared.

Zx::set():

```
void (*set) (struct calib_Zx_obj * rop,
const struct calib_Zx_obj * op);
```

Set `rop` to `op` in Zx, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to copy.

Zx::set_si():

```
void (*set_si) (struct calib_Zx_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in Zx, where:

`rop` is the polynomial receiving the result;
`op` is the integer value to set.

Zx::set_z():

```
void (*set_z) (struct calib_Zx_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in Zx, where:

`rop` is the polynomial receiving the result; and
`op` is the GMP integer value to set.

Zx::set_var_power():

```
void (*set_var_power)
 (struct calib_Zx_obj * rop,
  int power);
```

Set `rop` to `x ** power` in Zx, where:

`rop` is the polynomial receiving the result;
`power` is the power to set (must be non-negative).

Zx::add():

```
void (*add) (struct calib_Zx_obj *           rop,
const struct calib_Zx_obj * op1,
const struct calib_Zx_obj * op2);
```

Set `rop` to `op1 + op2` in Zx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zx::sub():

```
void (*sub) (struct calib_Zx_obj *           rop,
const struct calib_Zx_obj * op1,
const struct calib_Zx_obj * op2);
```

Set `rop` to `op1 - op2` in Zx, where:

`rop` is the polynomial receiving the result;

`op1` is the first operand; and
`op2` is the second operand.

Zx::neg():

```
void (*neg) (struct calib_Zx_obj *           rop,
const struct calib_Zx_obj * op);
```

Set `rop` to `- op` in Zx, where:

`rop` is the polynomial receiving the result;
`op` is the operand to negate.

Zx::mul():

```
void (*mul) (struct calib_Zx_obj *           rop,
const struct calib_Zx_obj * op1,
const struct calib_Zx_obj * op2);
```

Set `rop` to `op1 * op2` in Zx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zx::mul_z():

```
void (*mul_z) (struct calib_Zx_obj *         rop,
  const struct calib_Zx_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in Zx, where:

`rop` is the polynomial receiving the result;
`op1` is the first (polynomial) operand; and
`op2` is the second (GMP integer) operand.

Zx::ipow():

```
void (*ipow) (struct calib_Zx_obj * rop,
 const struct calib_Zx_obj * op,
 int power);
```

Set `rop` to `op ** power` in Zx, where: where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to exponentiate; and
`power` is the power to take (must be >= 0).

Zx::dup():

```
struct calib_Zx_obj *
(*dup) (const struct calib_Zx_obj * op);
```

Return a dynamically-allocated Zx polynomial that is a copy of `op`, where:

`op` is the polynomial to be duplicated.

Zx::free():

```
void (*free) (struct calib_Zx_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly` is the polynomial to be freed.

This is equivalent to performing `Zx -> clear (poly);`, followed by `free (poly);`.

Zx::eval():

```
void (*eval) (mpz_ptr rop,
 const struct calib_Zx_obj * poly,
 mpz_srcptr value);
```

Evaluate polynomial `poly` at the given `value`, storing the result in `rop`, where:

`rop` is the GMP integer receiving the result;
`poly` is the polynomial to be evaluated; and
`value` is the value at which to evaluate the polynomial.

Zx::div():

```
void (*div) (struct calib_Zx_obj * quotient,
struct calib_Zx_obj * remainder,
mpz_ptr d,
const struct calib_Zx_obj * a,
const struct calib_Zx_obj * b);
```

Polynomial pseudo-division in $Z[x]$, where:

`quotient` receives the quotient polynomial (may be NULL);
`remainder` receives the remainder polynomial (may be NULL);
`d` receives the "denominator" value (may be NULL);
`a` is the dividend polynomial; and
`b` is the divisor polynomial (must not be zero).

Pseudo-division has the following properties:

- $d * a = quotient * b + remainder$
- $degree(remainder) < degree(b)$

Zx::exactly_divides():

```
calib_bool
(*exactly_divides)
(struct calib_Zx_obj * quotient,
 const struct calib_Zx_obj * a,
 const struct calib_Zx_obj * b);
```

Return 1 if-and-only-if `a` is exactly divisible by `b` (setting `quotient` such that `a = b * quotient`); returns 0 otherwise (without modifying `quotient`), where:

`quotient` receives the quotient polynomial (may be NULL);
`a` is the dividend polynomial; and

`b` is the divisor polynomial (may not be zero).

Zx::div_z_exact():

```
void (*div_z_exact)
(struct calib_Zx_obj * rop,
 const struct calib_Zx_obj * op1,
 mpz_srcptr op2);
```

Set `rop` to `op1 / op2` in Zx, where:

`result` receives result polynomial;
`poly` is the dividend polynomial; and
`z` is the GMP integer by which to divide `poly`.

It is a *fatal error* if the division is not exact.

Zx::gcd():

```
void (*gcd) (struct calib_Zx_obj * gcd,
const struct calib_Zx_obj * a,
const struct calib_Zx_obj * b);
```

Compute the greatest common divisor (GCD) of `a` and `b`, storing the result in `gcd`, where:

`gcd` receives the resulting GCD polynomial;
`a` is the first operand polynomial; and
`b` is the second operand polynomial.

Zx::gcd_n():

```
void (*gcd_n) (struct calib_Zx_obj * gcd,
  struct calib_Zx_obj ** cofact,
  int npoly,
  const struct calib_Zx_obj * const *
poly_ptrs);
```

Compute the greatest common divisor (GCD) polynomial that simultaneously divides n given polynomials, where:

`gcd` receives the resulting GCD polynomial;
`cofact` is an array of n *pointers* to polynomials receiving the cofactor corresponding to each given input polynomial (may be NULL);
`npoly` is the number n of input polynomials provided; and
`poly_ptrs` is an array of n *pointers* to input polynomials whose GCD is to be computed.

This can be vastly more efficient that decomposing this into $n-1$ consecutive calls to the `gcd` function.

Zx::extgcd():

```
void (*extgcd) (struct calib_Zx_obj * gcd,
   struct calib_Zx_obj * xa,
   struct calib_Zx_obj * xb,
```

```
        const struct calib_Zx_obj * a,
        const struct calib_Zx_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`gcd`	receives the resulting GCD polynomial;
`xa`	receives the multiplier polynomial for `a`;
`xb`	receives the multiplier polynomial for `b`;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

The result satisfies the following properties:

1. $degree(xa) < degree(b)$
2. $degree(xb) < degree(a)$

Zx::prim_part():

```
void (*prim_part)
(mpz_ptr content,
 struct calib_Zx_obj * ppart,
 const struct calib_Zx_obj * op);
```

Compute the `content` and primitive part `ppart` of the given polynomial `op`, where:

`content`	a GMP integer receiving the content of `op`;
`ppart`	receives primitive part of `op`; and
`op`	the polynomial to be decomposed into content and primitive parts.

Zx::resultant():

```
void (*resultant)
(mpz_ptr result,
 const struct calib_Zx_obj * a,
 const struct calib_Zx_obj * b);
```

Compute the resultant of polynomials `a` and `b` (an integer value), storing the result in `result`, where:

`result`	is the resultant of given polynomials;
`a`	is the first operand; and
`b`	is the second operand.

Zx::discriminant():

```
void (*discriminant)
(mpz_ptr result
 const struct calib_Zx_obj * poly);
```

Compute the discriminant of polynomial `poly` (an integer value), storing the result in `result`, where:

`result`	is the discriminant of the given polynomial;
`poly`	is the polynomial whose discriminant is to be computed.

Zx::derivative():

```
void (*derivative)
(struct calib_Zx_obj * rop,
 const struct calib_Zx_obj * op);
```

Set `rop` to be the derivative of Zx polynomial `op`, where:

`rop` is the derivative of the given polynomial; and
`op` is the polynomial whose derivative is to be computed.

Zx::cvZpx():

```
struct calib_Zx_obj *
(*cvZpx) (const struct calib_Zpx_obj * op);
```

Return a dynamically-allocated polynomial in $Z[x]$ that corresponds to the given polynomial `op` in $Zp[x]$ (the Z coefficients chosen to have smallest absolute value that are equivalent to the corresponding Zp coefficient), where:

`op` is the Zpx polynomial to convert into signed Zx form.

Zx::factor():

```
struct calib_Zx_factor *
(*factor) (const struct calib_Zx_obj * poly);
```

Factor the given polynomial `poly` into its irreducible factors, returning a linked list of these factors, where:

`poly` is the Zx polynomial to be factored.

Note: If a suitable implementation of the Lenstra-Lenstra-Lovász (LLL) lattice-basis reduction algorithm is provided, this function uses the Van Hoeij algorithm that runs in polynomial time. (FPLLL is the only currently supported LLL implementation.) Otherwise the Zassenhaus algorithm is used that can require exponential time on certain classes of polynomials.

Zx::factor_square_free():

```
struct calib_Zx_factor *
(*factor_square_free)
(const struct calib_Zx_obj * poly);
```

Given a polynomial `poly` that must be both primitive and square-free (all factors occur exactly once), factor `poly` into its irreducible factors, returning a linked-list of these factors, where:

`poly` is the primitive, square-free Zx polynomial to be factored.

Note: See the above note regarding LLL and Van Hoeij algorithms.

Zx::finish_sqf():

```
struct calib_Zx_factor *
(*finish_sqf)
(const struct calib_Zx_factor * sqfactors);
```

Given a list of primitive, square-free polynomial `factors`, "finish" the factorization by factoring each such factor into irreducible polynomials, returning a linked-list of these factors, where:

`sqfactors` is a linked-list of primitive, square-free Zx polynomial factors for which factorization into irreducibles is desired.

Note: See the above note regarding LLL and Van Hoeij algorithms.

Zx::free_factors():

```
void (*free_factors)
(struct calib_Zx_factor * factors);
```

Free up the given list of Zx factors, where:

`factors` is a linked-list factors to be freed.

Zx::zerop():

```
calib_bool
(*zerop) (const struct calib_Zx_obj * op);
```

Return 1 if-and-only-if the given Zx polynomial is identically zero and 0 otherwise, where:

`op` is the Zx polynomial to test for zero.

Zx::onep():

```
calib_bool
(*onep) (const struct calib_Zx_obj * op);
```

Return 1 if-and-only-if the given Zx polynomial is identically one and 0 otherwise, where:

`op` is the Zx polynomial to test for one.

Zx::set_genrep():

```
void (*set_genrep) (struct calib_Zx_obj * rop,
       const struct calib_genrep * op,
       const char * var);
```

Convert the given genrep `op` into Zx form, interpreting `var` to be the name of the variable used by the Zx polynomial, storing the result into `rop`, where:

`rop` receives the resulting Zx polynomial;
`op` is the genrep to convert into Zx polynomial form; and
`var` is the variable name (appearing within genrep `op`) that is to be interpreted as the polynomial variable in Zx.

Zx::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Zx_obj * op,
      const char * var);
```

Return a dynamically-allocated genrep corresponding to the given Zx polynomial `op`, using `var` as the name of the polynomial variable within the returned genrep, where:

`op` is the Zx polynomial to convert into genrep form; and

`var` is the variable name to use in the genrep for the polynomial variable of Zx.

Zx::factors_to_genrep():

```
struct calib_genrep *
(*factors_to_genrep)
(const struct calib_Zx_factor * factors,
 const char * var);
```

Return a dynamically-allocated genrep corresponding to the given list of Zx `factors`, using `var` as the name of the polynomial variable within the returned genrep, where:

`factors` is the list of Zx polynomial factors to convert into genrep form; and

`var` is the variable name to use in the genrep for the polynomial variable of Zx.

Zx::print_maxima():

```
void (*print_maxima)
(const struct calib_Zx_obj * op);
```

Print the given Zx polynomial `op` to stdout using syntax that can be directly read by Maxima, where:

`op` is the Zx polynomial to be printed.

7 Qx — The Polynomial Ring $Q[x]$

CALIB provides the `Qx` domain, representing the ring $Q[x]$, the univariate polynomials having rational coefficients. The "values" of this domain are represented by the following object:

```
/*
 * An instance of a polynomial in Q[x].  We represent these as the product
 * of a rational factor with a primitive Z[x] polynomial whose leading
 * coefficient (if any) is strictly positive.
 */

struct calib_Qx_obj {
mpq_t qfact; /* Rational multiplier  */
int degree; /* Degree of polynomial  */
int size; /* Size of coeff[] array (degree < size) */
mpz_ptr coeff; /* Coefficients of polynomial.   */
};
```

These objects are subject to `init()` and `clear()` operations. All such objects must be initialized prior to use by any other CALIB operation. Memory leaks result if they are not cleared when done.

The following additional object is used to represent linked-lists of factors produced by various Qx factorization algorithms:

```
struct calib_Qx_factor {
int multiplicity;
struct calib_Qx_obj * factor;
struct calib_Qx_factor * next;
};
```

No data is needed to instantiate the `Qx` domain. (There is only one such object, and cannot be freed.) Nonetheless, the `Qx` domain object is the only way to access the operations provided by this domain.

One may access CALIB's `Qx` domain as follows:

```
#include "calib/Qx.h"
...
        const struct calib_Qx_dom * Qx;
struct calib_Qx_obj poly1, poly2;
...
Qx = calib_get_Qx_dom ();
...
Qx -> init (&poly1);
Qx -> init (&poly2);
...
Qx -> mul (&poly1, &poly1, &poly2);
...
Qx -> clear (&poly2);
Qx -> clear (&poly1);
```

The `struct calib_Qx_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Qx::init():

```
void (*init) (struct calib_Qx_obj * x);
```

Initialize the given Qx polynomial `x`, where:

`x` is the polynomial to initialize.

Qx::init_degree():

```
void (*init_degree) (struct calib_Qx_obj * x,
int degree);
```

Initialize the given Qx polynomial `x` (while assuring that internal buffers are sufficiently large to hold a polynomial of up to the given `degree` without further allocation), where:

`x` is the polynomial to initialize; and
`degree` is the guaranteed minimimum degree polynomial that `x` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Qx::alloc():

```
void (*alloc) (struct calib_Qx_obj * rop,
  int degree);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial of at least the given `degree`, where:

`rop` is the polynomial whose allocation is to be adjusted; and
`degree` is the guaranteed minimum degree polynomial that `rop` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Qx::clear():

```
void (*clear) (struct calib_Qx_obj * x);
```

Clear out the given polynomial `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x` is the polynomial to be cleared.

Qx::set():

```
void (*set) (struct calib_Qx_obj * rop,
const struct calib_Qx_obj * op);
```

Set `rop` to `op` in Qx, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to copy.

Qx::set_si():

```
void (*set_si) (struct calib_Qx_obj * rop,
```

```
        calib_si_t op);
```

Set `rop` to `op` in Qx, where:

`rop` is the polynomial receiving the result; and
`op` is the integer value to set.

Qx::set_z():

```
void (*set_z) (struct calib_Qx_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in Qx, where:

`rop` is the polynomial receiving the result; and
`op` is the GMP integer value to set.

Qx::set_q():

```
void (*set_q) (struct calib_Qx_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in Qx, where:

`rop` is the polynomial receiving the result; and
`op` is the GMP rational value to set.

Qx::set_var_power():

```
void (*set_var_power)
 (struct calib_Qx_obj * rop,
  int power);
```

Set `rop` to `x ** power` in Qx, where:

`rop` is the polynomial receiving the result;
`power` is the power to set (must be non-negative).

Qx::set_Zx():

```
void (*set_Zx) (struct calib_Qx_obj * rop,
   const struct calib_Zx_obj * op);
```

Set `rop` to `op` in Qx, where:

`rop` is the Qx polynomial receiving the result;
`op` is the source Zx polynomial.

Qx::add():

```
void (*add) (struct calib_Qx_obj *           rop,
const struct calib_Qx_obj * op1,
const struct calib_Qx_obj * op2);
```

Set `rop` to `op1 + op2` in Qx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and

op2 is the second operand.

Qx::sub():

```
void (*sub) (struct calib_Qx_obj *           result,
const struct calib_Qx_obj * a,
const struct calib_Qx_obj * b);
```

Set rop to op1 - op2 in Qx, where:

rop is the polynomial receiving the result;
op1 is the first operand; and
op2 is the second operand.

Qx::neg():

```
void (*neg) (struct calib_Qx_obj * rop,
const struct calib_Zxyz_obj * op);
```

Set rop to - op in Qx, where:

rop is the polynomial receiving the result;
op is the operand to negate.

Qx::mul():

```
void (*mul) (struct calib_Qx_obj *           rop,
const struct calib_Qx_obj * op1,
const struct calib_Qx_obj * op2);
```

Set rop to op1 * op2 in Qx, where:

rop is the polynomial receiving the result;
op1 is the first operand; and
op2 is the second operand.

Qx::mul_si():

```
void (*mul_si) (struct calib_Qx_obj *        rop);
   const struct calib_Qx_obj *  op1,
   calib_si_t op2);
```

Set rop to op1 * op2 in Qx, where:

rop is the polynomial receiving the result;
op1 is the first operand; and
op2 is the second operand.

Qx::mul_z():

```
void (*mul_z) (struct calib_Qx_obj *        rop);
   const struct calib_Qx_obj *  op1,
   mpz_ptr op2);
```

Set rop to op1 * op2 in Qx, where:

rop is the polynomial receiving the result;

`op1` is the first operand; and
`op2` is the second operand.

Qx::mul_q():

```
void (*mul_q) (struct calib_Qx_obj *        rop);
   const struct calib_Qx_obj *  op1,
   mpq_ptr op2);
```

Set `rop` to `op1 * op2` in Qx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Qx::ipow():

```
void (*ipow) (struct calib_Qx_obj * rop,
 const struct calib_Qx_obj * op,
 int power);
```

Set `rop` to `op ** power` in Qx, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to exponentiate; and
`power` is the power to take (must be >= 0).

Qx::dup():

```
struct calib_Qx_obj *
(*dup) (const struct calib_Qx_obj * op);
```

Return a dynamically-allocated Qx polynomial that is a copy of `op`, where:

`op` is the polynomial to be duplicated.

Qx::free():

```
void (*free) (struct calib_Qx_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly` is the polynomial to be freed.

This is equivalent to performing `Qx -> clear (poly);`, followed by `free (poly);`.

Qx::eval():

```
void (*eval) (mpq_ptr rop,
 const struct calib_Qx_obj * poly,
 mpq_srcptr value);
```

Evaluate polynomial `poly` at the given `value`, storing the result in `rop`, where:

`rop` is the GMP rational receiving the result;
`poly` is the polynomial to be evaluated; and
`value` is the value at which to evaluate the polynomial.

Qx::div():

```
void (*div) (struct calib_Qx_obj * quotient,
struct calib_Qx_obj * remainder,
const struct calib_Qx_obj * a,
const struct calib_Qx_obj * b);
```

Polynomial division in $Q[x]$, where:

`quotient`	receives the quotient polynomial (may be NULL);
`remainder`	receives the remainder polynomial (may be NULL);
`a`	is the dividend polynomial; and
`b`	is the divisor polynomial (may not be zero).

Division in Q[x] has the following properties:

- $a = quotient * b + remainder$
- $degree(remainder) < degree(b)$

Qx::gcd():

```
void (*gcd) (struct calib_Qx_obj * gcd,
const struct calib_Qx_obj * a,
const struct calib_Qx_obj * b);
```

Compute the greatest common divisor (GCD) of `a` and `b`, storing the result in `gcd`, where:

`gcd`	receives the resulting GCD polynomial (always monic);
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

Qx::extgcd():

```
void (*extgcd) (struct calib_Qx_obj * gcd,
   struct calib_Qx_obj * xa,
   struct calib_Qx_obj * xb,
   const struct calib_Qx_obj * a,
   const struct calib_Qx_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`gcd`	receives the resulting GCD polynomial (always monic);
`xa`	receives the multiplier polynomial for `a`;
`xb`	receives the multiplier polynomial for `b`;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

The result satisfies the following properties:

1. $degree(xa) < degree(b)$
2. $degree(xb) < degree(a)$

Qx::factor():

```
struct calib_Qx_factor *
(*factor) (const struct calib_Qx_obj * poly);
```

Factor the given polynomial `poly` into its irreducible factors, returning a linked list of these factors, where:

`poly` is the Qx polynomial to be factored.

Except for an optional leading constant factor, all other factors are monic and irreducible.

Note the discussion in Zx::factor() regarding Van Hoeij and LLL algorithms.

Qx::free_factors():

```
void (*free_factors)
(struct calib_Qx_factor * factors);
```

Free up the given list of Qx factors, where:

`factors` is a linked-list factors to be freed.

Qx::derivative():

```
void (*derivative)
(struct calib_Qx_obj * rop,
 const struct calib_Qx_obj * op);
```

Set `rop` to be the derivative of Qx polynomial `op`, where:

`rop` is the derivative of the given polynomial; and
`op` is the polynomial whose derivative is to be computed.

Qx::integral():

```
void (*integral)
(struct calib_Qx_obj * rop,
 const struct calib_Qx_obj * op);
```

Set `rop` to be the integral of Qx polynomial `op`, where:

`rop` is the derivative of the given polynomial; and
`op` is the polynomial whose integral is to be computed.

Qx::zerop():

```
calib_bool
(*zerop) (const struct calib_Qx_obj * op);
```

Return 1 if-and-only-if the given Qx polynomial is identically zero and 0 otherwise, where:

`op` is the Qx polynomial to test for zero.

Qx::onep():

```
calib_bool
(*onep) (const struct calib_Qx_obj * op);
```

Return 1 if-and-only-if the given Qx polynomial is identically one and 0 otherwise, where:

`op` is the Qx polynomial to test for one.

Qx::set_genrep():

```
void (*set_genrep) (struct calib_Qx_obj * rop,
        const struct calib_genrep * op,
        const char * var);
```

Compute a Qx polynomial obtained from the given genrep `op`, interpreting `var` to be the name of the variable used by the Qx polynomial, storing the result in `rop`, where:

`rop` receives the resulting Qx polynomial;
`op` is the genrep to convert into Qx polynomial form; and
`var` is the variable name (appearing within genrep `op`) that is to be interpreted as the polynomial variable in Qx.

Qx::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Qx_obj * op,
       const char * var);
```

Return a dynamically-allocated genrep corresponding to the given Qx polynomial `op`, using `var` as the name of the polynomial variable within the returned genrep, where:

`op` is the Qx polynomial to convert into genrep form; and
`var` is the variable name to use in the genrep for the polynomial variable of Qx.

Qx::factors_to_genrep():

```
struct calib_genrep *
(*factors_to_genrep)
(const struct calib_Qx_factor * factors,
 const char * var);
```

Return a dynamically-allocated genrep corresponding to the given list of Qx `factors`, using `var` as the name of the polynomial variable within the returned genrep, where:

`factors` is the list of Qx polynomial factors to convert into genrep form; and
`var` is the variable name to use in the genrep for the polynomial variable of Qx.

Qx::get_coeffs():

```
mpq_ptr (*get_coeffs) (const struct calib_Qx_obj * op);
```

Return a dynamically allocated array of GMP rationals representing the coefficients of Qx polynomial `op`, where:

`op` is the Qx polynomial for which to get the coefficients.

It is the caller's responsibility to free the returned array (of length `op->degree + 1`).

Qx::set_coeffs():

```
void (*set_coeffs) (struct calib_Qx_obj * rop,
        int degree,
        mpq_srcptr coeffs);
```

Set `rop` to be the $Q[x]$ polynomial having the given degree and rational coefficients, where:

`rop`	is the Qx polynomial receiving the result;
`degree`	is the degree to set; and
`coeffs`	is the array of GMP rational coefficients (of length `degree + 1`) to set.

8 Zxyz — The Polynomial Ring $Z[x, y, z, ...]$

CALIB provides the `Zxyz` domain, representing the ring $Z[x, y, z, ...]$, the multivariate polynomials having integer coefficients. The "values" of this domain are represented by the following object:

```
struct calib_Zxyz_obj {
int nterms; /* Number of terms in polynomial */
int size; /* Size of pow[] and coeff[] arrays */
const struct calib_Zxyz_dom *
dom; /* Domain of polynomial */
int * pow; /* Powers of each var, each term */
mpz_ptr coeff; /* Coefficient for each term */
};
```

A Zxyz polynomial with n terms and k variables uses $n*k$ elements of the `pow` array (each term specifies the exponent for all k variables); and n elements of the `coeff` array. The terms are sorted lexically by their k-element power vectors (largest degrees before smaller).

These objects are subject to `init()` and `clear()` operations. All such objects must be initialized prior to use by any other CALIB operation. Memory leaks result if they are not cleared when done.

The following additional object is used to represent linked-lists of factors produced by various Zxyz factorization algorithms:

```
struct calib_Zxyz_factor {
int multiplicity;
struct calib_Zxyz_obj * factor;
struct calib_Zxyz_factor * next;
};
```

When making a `Zxyz` domain, one must specify the number of variables and a textual name for each. These are stored in the fields:

```
struct calib_Zxyz_dom {
int nvars; /* Number of variables */
char ** vars; /* Name of each variable */
...
};
```

One may access CALIB's `Zxyz` domain as follows:

```
#include "calib/Zxyz.h"
...
        const struct calib_Zxyz_dom * Zxyz;
struct calib_Zxyz_obj poly1, poly2;
const char * varnames [3] = {"x", "y", "z"};
...
Zxyz = calib_make_Zxyz_dom (3, varnames);
...
Zxyz -> init (&poly1);
Zxyz -> init (&poly2);
...
```

```
Zxyz -> mul (&poly1, &poly1, &poly2);
...
Zxyz -> clear (&poly2);
Zxyz -> clear (&poly1);
...
calib_free_Zxyz_dom (Zxyz);
```

The `struct calib_Zxyz_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Zxyz::init():

```
void (*init) (const struct calib_Zxyz_dom * dom,
 struct calib_Zxyz_obj * x);
```

Initialize the given Zxyz polynomial `x`, where:

`dom` is the Zxyz ring/domain the polynomial belongs to; and
`x` is the polynomial to initialize.

Zxyz::init_si():

```
void (*init_si) (const struct calib_Zxyz_dom * dom,
    struct calib_Zxyz_obj * x,
    calib_si_t         op);
```

Initialize the given Zxyz polynomial `x` to the given constant value `op`, where:

`dom` is the Zxyz ring/domain the polynomial belongs to; and
`x` is the polynomial to initialize; and
`op` is the constant value to which polynomial `x` is set.

Zxyz::alloc():

```
void (*alloc) (struct calib_Zxyz_obj * rop,
  int nterms);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial having at least the given `nterms` number of (non-zero) terms, where:

`rop` is the polynomial whose allocation is to be adjusted; and
`nterms` is the guaranteed minimum number of terms that polynomial `rop` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Zxyz::clear():

```
void (*clear) (struct calib_Zxyz_obj * x);
```

Clear out the given polynomial `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x` is the polynomial to be cleared.

Zxyz::set():

```
void (*set) (struct calib_Zxyz_obj * rop,
const struct calib_Zxyz_obj * op);
```

Set `rop` to `op` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to copy.

Zxyz::set_si():

```
void (*set_si) (struct calib_Zxyz_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op` is the integer value to set.

Zxyz::set_z():

```
void (*set_z) (struct calib_Zxyz_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op` is the GMP integer value to set.

Zxyz::set_var_power():

```
void (*set_var_power)
 (struct calib_Zxyz_obj * rop,
  int var,
  int power);
```

Set `rop` to `var ** power` in Zxyz, where:

`rop` is the polynomial receiving the result;
`var` is the index of the variable; and
`power` is the power to set (must be non-negative).

Zxyz::add():

```
void (*add) (struct calib_Zxyz_obj * rop,
const struct calib_Zxyz_obj * op1,
const struct calib_Zxyz_obj * op2);
```

Set `rop` to `op1 + op2` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zxyz::add_n():

```
void (*add_n) (struct calib_Zxyz_obj * rop,
  int npoly,
  const struct calib_Zxyz_obj ** poly_ptrs);
```

Add `npoly` polynomials together (given by the `poly_ptrs`), storing the result in `rop`, where:

`rop` is the polynomial receiving the result;
`npoly` is the number of polynomials being added; and
`poly_ptrs` is an array of *pointers* to the polynomials being added.

Zxyz::sub():

```
void (*sub) (struct calib_Zxyz_obj * rop,
const struct calib_Zxyz_obj * op1,
const struct calib_Zxyz_obj * op2);
```

Set `rop` to `op1 - op2` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zxyz::neg():

```
void (*neg) (struct calib_Zxyz_obj * rop,
const struct calib_Zxyz_obj * op);
```

Set `rop` to `- op` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op` is the operand to negate.

Zxyz::mul():

```
void (*mul) (struct calib_Zxyz_obj * rop,
const struct calib_Zxyz_obj * op1,
const struct calib_Zxyz_obj * op2);
```

Set `rop` to `op1 * op2` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zxyz::mul_z():

```
void (*mul_z) (struct calib_Zxyz_obj * rop,
  const struct calib_Zxyz_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op1` is the first (polynomial) operand; and
`op2` is the second (GMP integer) operand.

Zxyz::ipow():

```
void (*ipow) (struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * op,
 int power);
```

Set `rop` to `op ** power` in Zxyz, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to exponentiate; and
`power` is the power to take (must be >= 0).

Zxyz::dup():

```
struct calib_Zxyz_obj *
(*dup) (const struct calib_Zxyz_obj * op);
```

Return a dynamically-allocated Zxyz polynomial that is a copy of `op`, where:

`op` is the polynomial to be duplicated.

Zxyz::free():

```
void (*free) (struct calib_Zxyz_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly` is the polynomial to be freed.

This is equivalent to performing `Zxyz -> clear (poly);`, followed by `free (poly);`.

Zxyz::eval():

```
void (*eval) (mpz_ptr rop,
 const struct calib_Zxyz_obj * poly,
 mpz_srcptr values);
```

Evaluate polynomial `poly` at the given `values`, storing the result in `rrop`, where:

`rop` is the GMP integer receiving the result;
`poly` is the polynomial to be evaluated; and
`values` is an array of GMP integer values (one per variable) at which to evaluate the polynomial.

Zxyz::eval_var_subset():

```
void (*eval_var_subset)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * poly,
 const calib_bool * evflags,
 mpz_srcptr values);
```

Evaluate polynomial `poly` at a specified *subset* of its variables, storing the result in `rop`. For each variable `i`, evaluate away variable `i` at value `values[i]` if-and-only-if `evflags[i]` is true, where:

`rop` is the GMP integer receiving the result;
`poly` is the polynomial to be evaluated;
`evflags` is an array of booleans (one per variable) for which TRUE means to evaluate the corresponding variable; and
`values` is an array of GMP integer values (one per variable) at which to evaluate the polynomial.

Zxyz::div():

```
void (*div) (struct calib_Zxyz_obj * quotient,
```

```
struct calib_Zxyz_obj * remainder,
struct calib_Zxyz_obj * d,
const struct calib_Zxyz_obj * a,
const struct calib_Zxyz_obj * b,
int var);
```

Polynomial pseudo-division in $Z[x, y, z]$ with respect to a specified main variable `var`, where:

`quotient`	receives the quotient polynomial (may be NULL);
`remainder`	receives the remainder polynomial (may be NULL);
`d`	receives the "denominator" value (may be NULL);
`a`	is the dividend polynomial;
`b`	is the divisor polynomial (may not be zero);
`var`	is the main variable for division.

Pseudo-division has the following properties:

- $d * a = quotient * b + remainder$
- $degree(remainder, var) < degree(b, var)$

Zxyz::div_z_exact():

```
void (*div_z_exact)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * op1,
 mpz_srcptr op2);
```

Set `rop` to `op1 / op2` in Zxyz, where:

`rop`	receives result polynomial;
`op1`	is the dividend polynomial; and
`op2`	is the GMP integer by which to divide `poly`.

It is a *fatal error* if the division is not exact.

Zxyz::div_remove():

```
int (*div_remove)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * op1,
 const struct calib_Zxyz_obj * op2);
```

Repeatedly divide polynomial `op1` by polynomial `op2` until no more factors `op2` can be removed, storing `op1 / op2**k` into `rop` and returning `k`, where:

`rop`	receives result polynomial;
`op1`	is the dividend polynomial; and
`op2`	is the divisor polynomial.

Zxyz::gcd():

```
void (*gcd) (struct calib_Zxyz_obj * gcd,
const struct calib_Zxyz_obj * a,
const struct calib_Zxyz_obj * b);
```

Compute the greatest common divisor (GCD) of `a` and `b`, storing the result in `gcd`, where:

`gcd`	receives the resulting GCD polynomial;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

Zxyz::gcd_n():

```
void (*gcd_n) (struct calib_Zxyz_obj * gcd,
  struct calib_Zxyz_obj ** cofact,
  int npoly,
  const struct calib_Zxyz_obj **
array);
```

Compute the greatest common divisor (GCD) polynomial that simultaneously divides n given polynomials, where:

`gcd`	receives the resulting GCD polynomial;
`cofact`	is an array of `nfact` *pointers* to polynomials receiving the cofactor corresponding to each given input polynomial (may be NULL);
`npoly`	is the number of input polynomials provided; and
`array`	is an array of `nfact` *pointers* to input polynomials whose GCD is to be computed.

This can be vastly more efficient that decomposing this into $nfact - 1$ consecutive calls to the `gcd` function.

Zxyz::extgcd():

```
void (*extgcd) (struct calib_Zxyz_obj * gcd,
   struct calib_Zxyz_obj * xa,
   struct calib_Zxyz_obj * xb,
   mpz_ptr d,
   const struct calib_Zxyz_obj * a,
   const struct calib_Zxyz_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`gcd`	receives the resulting GCD polynomial;
`xa`	receives the multiplier polynomial for `a`;
`xb`	receives the multiplier polynomial for `b`;
`d`	receives the denominator for `xa` and `xb`;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

Zxyz::z_content():

```
void (*z_content)
(mpz_ptr zcont,
 const struct calib_Zxyz_obj * op);
```

Compute the integer content `zcont` of the given polynomial `op`, where:

`zcont`	a GMP integer receiving the content of `op`;
`op`	the polynomial whose integer content is to be computed.

Zxyz::strip_z_content():

```
void (*strip_z_content)
(mpz_ptr zcont,
 struct calib_Zxyz_obj * poly);
```

Compute and remove the `content` from the given polynomial `poly`, where:

`zcont`	a GMP integer receiving the content of `poly`;
`poly`	the polynomial whose content is to be computed and removed.

Zxyz::prim_part():

```
void (*prim_part)
(mpz_ptr content,
 struct calib_Zxyz_obj * ppart,
 const struct calib_Zxyz_obj * op,
 int var);
```

Compute the multi-variate polynomial `content` (with respect to given main variable `var`) of polynomial `op` and setting `ppart` to be the primitive part, where:

`content`	receives the content of `op` with respect to variable `var`;
`ppart`	receives primitive part of `op` with respect to variable `var`;
`op`	the polynomial to be decomposed into content and primitive parts; and
`var`	is the main variable with respect to which the content is computed.

Zxyz::resultant():

```
void (*resultant)
(mpz_ptr result,
 const struct calib_Zxyz_obj * a,
 const struct calib_Zxyz_obj * b,
 int var);
```

Compute the resultant of polynomials `a` and `b` with respect to given main variable `var`, storing the result in `result`, where:

`result`	is the resultant of given polynomials;
`a`	is the first operand;
`b`	is the second operand; and
`var`	is the main variable to use / eliminate.

Zxyz::resultant_old():

```
struct calib_Zxyz_factor *
(*resultant)
(const struct calib_Zxyz_obj * a,
 const struct calib_Zxyz_obj * b,
 int var);
```

Compute the resultant of polynomials `a` and `b` with respect to given main variable `var`, returning the result as a partially-factored list of factors, where:

`a`	is the first operand;
`b`	is the second operand; and
`var`	is the main variable to use / eliminate.

Note: This is an old and very naive implementation!!! Use only on small polynomials of fairly low degree!

Zxyz::discriminant():

```
void (*discriminant)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * op,
 int var);
```

Set `rop` to be the discriminant of polynomial `op` with respect to variable `var`, where:

`rop`	is the resulting polynomial;
`op`	is the polynomial for which to compute the discriminant; and
`var`	is the main variable to use / eliminate.

Zxyz::cvZpxyz():

```
void (*cvZpxyz) (const calib_Zxyz_obj * rop,
    const struct calib_Zp_dom * Zp,
    const struct calib_Zxyz_obj * op);
```

Compute a polynomial in $Z[x, y, z]$ that is the corresponding representative of the given polynomial `op` in $Zp[x, y, z]$ (given in Zxyz form but having coefficients mod p — the Z coefficients are chosen to have smallest absolute value that are equivalent to the correponding Zp coefficient), storing the result in `rrop`, where:

`rop`	receives the resulting polynomial;
`Zp`	is the source Zp coefficient domain; and
`op`	is the Zxyz polynomial (with Zp coefficients) to convert into Zxyz form.

Zxyz::factor():

```
struct calib_Zxyz_factor *
(*factor) (const struct calib_Zxyz_obj * poly);
```

Factor the given polynomial `poly` into its irreducible factors, returning a linked list of these factors, where:

`poly`	is the Zxyz polynomial to be factored.

See the Note in Zx::factor() regarding the Van Hoeij and LLL algorithms.

Zxyz::sqf_factor():

```
struct calib_Zxyz_factor *
(*sqf_factor)
(const struct calib_Zxyz_obj * poly);
```

Perform “square-free factorization” of given polynomial `poly`, where:

`poly` is the Zxyz polynomial to be square-free factored.

Zxyz::free_factors():

```
void (*free_factors)
(struct calib_Zxyz_factor * factors);
```

Free up the given list of Zxyz factors, where:

`factors` is a linked-list factors to be freed.

Zxyz::map_to_subring():

```
struct calib_Zxyz_obj *
(*map_to_subring)
(struct calib_Zxyz_obj * result,
 const struct calib_Zxyz_obj * poly,
 const struct calib_Zxyz_dom * subring,
 const int * varmap);
```

Map the given polynomial `poly` from its original ring into a new polynomial whose coefficients reside in the given subring. The `varmap` array controls this mapping on a variable-by-variable basis. Let i be a variable index within the original polynomial ring, and let $j = varmap[i]$. Then $j = -1$ ==> variable i remains in the source ring, whereas $j >= 0$ ==> variable i (from the original ring) maps to variable j of the subring. (j must satisfy $0 <= j < subring.nvars$.)

Since the calib_Zxyz_obj representation handles only GMP integer coefficients, the `result` object contains only dummy "1" coefficient values. The *actual* coefficients are found in this function's return values, which is an array of `struct calib_Zxyz_obj` objects (each of whose `dom` member is the given subring). This returned array has the same number of elements as `result -> nterms`. It is the caller's responsibility to free up the coefficient array when done with it. The arguments are:

`result` The high-level structural result (but with "dummy" coefficient values of 1);

`poly` is the Zxyz polynomial to be mapped into the given `subring`;

`subring` is the Zxyz domain describing the subring into which we are mapping coefficients; and

`varmap` is the array specifying how each variable in `poly` is to be mapped.

Zxyz::copy_into_superring():

```
struct calib_Zxyz_obj *
(*copy_into_superring)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * op);
```

Set `rop` to `op`. The domain of `rop` and `op` need not be the same, but the `rop` domain must have at least as many variables as the domain of `op`. Extra variables receive a power of zero in every term copied. The arguments are:

`rop` the destination polynomial / domain; and `op` the source polynomial / domain.

Zxyz::convert_with_varmap():

```
void (*convert_with_varmap)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zxyz_obj * op,
 const int * varmap);
```

Convert a polynomial from one ring to another, mapping variables according to the given `varmap`. Variables mapping to -1 in the `varmap` must not appear in the source polynomial `op`. The arguments are:

`rop` the destination polynomial / domain;
`op` the source polynomial / domain; and
`varmap` the variable mapping array.

Zxyz::copy_from_Zx():

```
void (*copy_from_Zx)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zx_obj * op,
 int var);
```

Set Zxyz polynomial `rop` to Zx polynomial `op`. The polynomial `op` becomes a polynomial in the given `var` of the `rop` polynomial. The arguments are:

`rop` the destination Zxyz polynomial / domain;
`op` the source Zx polynomial; and
`var` the `dst` polynomial variable to use.

Zxyz::copy_into_Zx():

```
void (*copy_into_Zx)
(struct calib_Zx_obj * rop,
 const struct calib_Zxyz_obj * op,
 int var);
```

Set Zx polynomial `rop` to Zxyz polynomial `op` (which must contain only given variable `var`), where:

`rop` the destination Zx polynomial;
`op` the source Zxyz polynomial / domain; and
`var` the source polynomial variable.

Zxyz::copy_from_Zpx():

```
void (*copy_from_Zpx)
(struct calib_Zxyz_obj * rop,
 const struct calib_Zpx_obj * op,
 int var);
```

Set Zxyz polynomial `rop` from Zpx polynomial `op`. The polynomial `op` becomes a polynomial in the given `var` of the `rop` polynomial. The arguments are:

`rop` the destination Zxyz polynomial / domain;
`op` the source Zpx polynomial; and
`var` the destination polynomial variable to use.

Zxyz::copy_into_Zpx():

```
void (*copy_into_Zpx)
(struct calib_Zpx_obj * rop,
 const struct calib_Zxyz_obj * op,
 int var);
```

Set Zpx polynomial `rop` to Zxyz polynomial `op` (which must contain only givenvariable `var`), where:

`rop` the destination Zpx polynomial;
`op` the source Zxyz polynomial / domain; and
`var` the source polynomial variable.

Zxyz::copy_from_Qax():

```
void (*copy_from_Qax)
(struct calib_Zxyz_obj * rop,
 mpz_ptr denom,
 const struct calib_Qax_obj * op,
 int xvar,
 int avar);
```

Copy the given `op` polynomial (in Qax form) into the given destination polynomial (in Zxyz form). The `op` polynomial's main variable becomes `xvar` of the `rop` polynomial, while the `op` polynomial's algebraic variable becomes the given `avar` of the `rop` polynomial, where:

`rop` the destination Zxyz polynomial / domain;
`denom` receives the common denominator;
`op` the source Qax polynomial;
`xvar` main variable of `op` becomes variable `xvar` in `rop` polynomial; and
`avar` algebraic variable of `op` becomes variable `avar` in `rop` polynomial.

Zxyz::copy_into_Qax():

```
void (*copy_into_Qax)
(struct calib_Qax_obj * rop,
 const struct calib_Zxyz_obj * op,
 int xvar,
 int avar);
```

Set Qax polynomial `rop` to Zxyz polynomial `op`. Variable `xvar` of `op` becomes the main variable of `rop`, while variable `avar` of `op` becomes the algebraic variable of `rop`, where:

`rop` the destination Qax polynomial;
`op` the source Zxyz polynomial / domain; and
`xvar` source polynomial variable `xvar` becomes the main variable of `rop`; and
`avar` source polynomial variable `avar` becomes the algebraic number variable of `rop`.

Zxyz::add_vars():

```
struct calib_Zxyz_dom *
(*add_vars)
(const struct Zxyz_dom * dom,
 int nvars,
 const char * const * newvars);
```

Create a new Zxyz domain having the given additional variables over those in given domain `dom`, where:

`dom`	is the original Zxyz domain;
`nvars`	is the number of variables to add; and
`newvars`	is an array of the new variable names being added.

Zxyz::zerop():

```
calib_bool
(*zerop) (const struct calib_Zxyz_obj * op);
```

Return 1 if-and-only-if the given Zxyz polynomial is identically zero and 0 otherwise, where:

`op`	is the Zxyz polynomial to test for zero.

Zxyz::onep():

```
calib_bool
(*onep) (const struct calib_Zxyz_obj * op);
```

Return 1 if-and-only-if the given Zxyz polynomial is identically 1 and 0 otherwise, where:

`op`	is the Zxyz polynomial to test for one.

Zxyz::set_genrep():

```
void (*set_genrep) (struct calib_Zxyz_obj * rop,
        const struct calib_genrep * op);
```

Compute a Zxyz polynomial obtained from the given genrep `op`, storing the result in `rop`. Use the domain of `rop` to map variable names in `op` to variable numbers in the resulting polynomial, where:

`rop`	receives the resulting Zxyz polynomial;
`op`	is the genrep to convert into Zxyz polynomial form.

Zxyz::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Zxyz_obj * op);
```

Return a dynamically-allocated genrep corresponding to the given Zxyz polynomial `op` (whose Zxyz domain provides variable names to use for each variable number), where:

`op`	is the Zxyz polynomial to convert into genrep form.

Zxyz::factors_to_genrep():

```
struct calib_genrep *
```

```
(*factors_to_genrep)
(const struct calib_Zxyz_factor * factors);
```

Return a dynamically-allocated genrep corresponding to the given list of Zxyz `factors` (each of whose Zxyz domain provides variable names to use for each variable number), where:

`factors` is the list of Zxyz polynomial factors to convert into genrep form.

Zxyz::print_maxima():

```
void (*print_maxima)
(const struct calib_Zxyz_obj * op);
```

Print the given Zxyz polynomial `op` to stdout using syntax that can be directly read by Maxima, where:

`op` is the Zxyz polynomial to be printed.

Zxyz::print_maxima_nnl():

```
void (*print_maxima_nnl)
(const struct calib_Zxyz_obj * op);
```

Print the given Zxyz polynomial `op` to stdout using syntax that can be directly read by Maxima (but with no terminating newline), where:

`op` is the Zxyz polynomial to be printed.

Zxyz::lookup_var():

```
int (*lookup_var)
(const struct calib_Zxyz_dom * dom,
 const char * var);
```

Return the index of the variable whose name is `var`, or -1 if `var` does not match any of the variables in the given domain `dom`, where:

`dom` is the Zxyz domain within which to lookup variable `var`; and
`var` is the variable name to query.

9 Zp — The Integers Modulo a Prime p

CALIB provides the `Zp` domain, representing the field of integers modulo a given prime p. The `Zp` domain does not provide a separate "value object," but rather represents `Zp` values directly using GMP integers. Because GMP integers do not have anywhere to record the CALIB domain to which they belong, most `Zp` operations require that the particular `Zp` domain be passed as an argument (usually the first argument), so that the particular prime `p` is available for computing over integers modulo `p`.

One may access CALIB's `Zp` domain as follows:

```
#include "calib/Zp.h"
...
mpz_t big_prime;
struct calib_Zp_dom * Zp_23;
struct calib_Zp_dom * Zp_big;

Zp_23 = calib_make_Zp_dom_ui (23);

mpz_init_set_ui (big_prime, 18446744073709551557);
Zp_big = calib_make_Zp_dom (big_prime);
...
calib_Zp_free_dom (Zp_big);
calib_Zp_free_dom (Zp_23);
```

The `struct calib_Zp_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Zp::set_si():

```
void (*set_si) (const struct calib_Zp_dom * f,
   mpz_ptr rop,
   calib_si_t op);
```

Set `rop` to `op` in Zp (`op` is reduced modulo `p` before assigning to `rop`), where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result; and
`op` is the source operand.

Zp::set_z():

```
void (*set_z) (const struct calib_Zp_dom * f,
  mpz_ptr rop,
  mpz_srcptr op);
```

Set `rop` to `op` in Zp (`op` is reduced modulo `p` before assigning to `rop`), where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result; and
`op` is the source operand.

Zp::set_q():

```
void (*set_q) (const struct calib_Zp_dom * f,
```

```
mpz_ptr rop,
mpq_srcptr op);
```

Set `rop` to `op` in Zp (`op` is reduced modulo `p` before assigning to `rop`), where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result; and
`op` is the source GMP rational operand.

Zp::add():

```
void (*add) (const struct calib_Zp_dom * f,
mpz_ptr rop,
mpz_srcptr op1,
mpz_srcptr op2);
```

Set `rop` to `op1 + op2` in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zp::sub():

```
void (*sub) (const struct calib_Zp_dom * f,
mpz_ptr rop,
mpz_srcptr op1,
mpz_srcptr op2);
```

Set `rop` to `op1 - op2` in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zp::neg():

```
void (*neg) (const struct calib_Zp_dom * f,
mpz_ptr rop,
mpz_srcptr op);
```

Set `rop` to `- op` in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result; and
`op` is the source operand.

Zp::mul():

```
void (*mul) (const struct calib_Zp_dom * f,
mpz_ptr rop,
mpz_srcptr op1,
mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zp::ipow():

```
void (*ipow) (const struct calib_Zp_dom * f,
 mpz_ptr rop,
 mpz_srcptr op,
 int power);
```

Set `rop` to `op ** power` in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result;
`op` is the operand to exponentiate; and
`power` is the power to take.

The exponent `power` is allowed to be any integer (positive, negative, or zero).

Zp::inv():

```
void (*inv) (const struct calib_Zp_dom * f,
mpz_ptr rop,
mpz_srcptr op);
```

Set `rop` to the multiplicative inverse of `op` in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result; and
`op` is the source operand (must not be zero).

Zp::set_random():

```
void (*set_random) (const struct calib_Zp_dom * f,
        mpz_ptr rop,
        struct calib_Random * randp);
```

Set `rop` to be a randomly selected value in Zp, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result; and
`randp` is the random number generator to use.

Zp::set_genrep():

```
void (*set_genrep) (const struct calib_Zp_dom * f,
        mpz_ptr rop,
        const struct calib_genrep * op);
```

Converts operand `op` (in "genrep" form that must be either an integer or rational constant) into the corresponding member of Zp, storing the result in `rop`, where:

`f` is the Zp domain performing this operation;
`rop` is the GMP integer receiving the result;
`op` is the source genrep operand.

Zp::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Zp_dom * f,
     mpz_srcptr op);
```

Converts operand `op` into the corresponding member of Zp, returning the result as a dynamically-allocated “genrep”, where:

`f` is the Zp domain performing this operation;
`op` is the source operand.

10 Zpx — The Polynomial Ring $Zp[x]$

CALIB provides the `Zpx` domain, representing the ring $Zp[x]$, the univariate polynomials having coefficients that are integers modulo prime p. The "values" of this domain are represented by the following object:

```
struct calib_Zpx_obj {
int degree; /* Degree of polynomial  */
int size; /* Size of coeff[] array (degree < size) */
const struct calib_Zpx_dom *
dom; /* Polynomial domain.   */
mpz_ptr coeff; /* Coefficients of polynomial.   */
};
```

The following additional object is used to represent linked-lists of factors produced by various Zpx factorization algorithms:

```
struct calib_Zpx_factor {
int multiplicity;
struct calib_Zpx_obj * factor;
struct calib_Zpx_factor * next;
};
```

The CALIB `Zpx` domain is constructed by specifying a `Zp` domain used to represent the coefficients of the corresponding Zpx polynomials.

One may access CALIB's `Zpx` domain as follows:

```
#include "calib/Zpx.h" /* #includes "calib/Zp.h" */
...
struct calib_Zp_dom * Zp;
        struct calib_Zpx_dom * Zpx;
struct calib_Zpx_obj poly1, poly2;
...
Zp  = calib_get_Zp_dom_ui (47); /* Integers modulo 47 */
Zpx = calib_make_Zpx_dom (Zp);
...
Zpx -> init (&poly1);
Zpx -> init (&poly2);
...
Zpx -> mul (&poly1, &poly1, &poly2);
...
Zpx -> clear (&poly2);
Zpx -> clear (&poly1);
calib_free_Zpx_dom (Zpx);
calib_free_Zp_dom (Zp);
```

The CALIB `Zpx` domain supports the following settings:

```
/*
 * Which factorization algorithm to use (for square-free polynomials).
 */
```

```
enum calib_Zpx_factor_method {
CALIB_ZPX_FACTOR_METHOD_BERLEKAMP,
CALIB_ZPX_FACTOR_METHOD_DDF,
};

/*
 * The "settings" object for the Zpx domain.
 */

struct calib_Zpx_settings {
/* Factorization method for square-free Zpx polynomials. */
enum calib_Zpx_factor_method factor_method;
};

/*
 * Newly created Zpx domains default to these settings.
 */

extern struct calib_Zpx_settings calib_Zpx_default_settings;
```

Each CALIB `Zpx` domain has its own copy of these settings (consulted by the domain's operations):

```
struct calib_Zpx_dom {
...
struct calib_Zpx_settings settings;
...
};
```

These settings are initialized from `calib_Zpx_default_settings` when the domain is constructed, but applications may alter these settings after construction, if desired.

The default `Zpx` `factor_method` is to use Berlekamp's algorithm.

The `struct calib_Zpx_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Zpx::init():

```
void (*init) (const struct calib_Zpx_dom * K_of_x,
 struct calib_Zpx_obj * x);
```

Initialize the given Zpx polynomial `x`, where:

`K_of_x`	is the Zpx ring/domain the polynomial belongs to; and
`x`	is the polynomial to initialize.

Zpx::init_degree():

```
void (*init_degree)
(const struct calib_Zpx_dom * K_of_x,
 struct calib_Zpx_obj * x,
 int degree);
```

Initialize the given Zpx polynomial `x` (while assuring that internal buffers are sufficiently large to hold a polynomial of up to the given `degree` without further allocation), where:

`K_of_x` is the Zpx ring/domain the polynomial belongs to;
`x` is the polynomial to initialize; and
`degree` is the guaranteed minimimum degree polynomial that `x` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Zpx::alloc():

```
void (*alloc) (struct calib_Zpx_obj * rop,
  int degree);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial of at least the given `degree`, where:

`rop` is the polynomial whose allocation is to be adjusted; and
`degree` is the guaranteed minimum degree polynomial that `rop` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Zpx::clear():

```
void (*clear) (struct calib_Zpx_obj * x);
```

Clear out the given polynomial `dst` (freeing all memory it might hold and returning it to the constant value of zero), where:

`dst` is the polynomial to be cleared.

Zpx::set():

```
void (*set) (struct calib_Zpx_obj * rop,
const struct calib_Zpx_obj * op);
```

Set `rop` to `op` in Zpx, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to copy.

Zpx::set_si():

```
void (*set_si) (struct calib_Zpx_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in Zpx, where:

`rop` is the polynomial receiving the result;
`op` is the integer value to set.

Zpx::set_z():

```
void (*set_z) (struct calib_Zpx_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in Zpx, where:

`rop` is the polynomial receiving the result;

`op` is the GMP integer value to set.

Zpx::set_q():

```
void (*set_q) (struct calib_Zpx_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in Zpx, where:

`rop` is the polynomial receiving the result;
`op` is the GMP rational value to set.

Zpx::set_var_power():

```
void (*set_var_power)
 (struct calib_Zpx_obj * rop,
  int power);
```

Set `rop` to `x ** power` in Zpx, where:

`rop` is the Zpx polynomial receiving the result;
`power` is the power to set (must be non-negative).

Zpx::set_Zx():

```
void (*set_Zx) (struct calib_Zpx_obj * rop,
   const struct calib_Zx_obj * op);
```

Set `rop` to `op` in Zpx, where:

`rop` is the destination Zpx polynomial; and
`op` is the source Zx polynomial.

The coefficients of the source polynomial are reduced modulo p during the copy.

Zpx::set_Qa():

```
void (*set_Qa) (struct calib_Zpx_obj * rop,
   const struct calib_Qa_obj * op);
```

Set `rop` to `op` in Zpx, where:

`rop` is the destination Zpx polynomial; and
`op` is the source Qa polynomial.

The rational coefficients of the source polynomial are reduced modulo p during the copy.

Zpx::add():

```
void (*add) (struct calib_Zpx_obj * rop,
const struct calib_Zpx_obj * op1,
const struct calib_Zpx_obj * op2);
```

Set `rop` to `op1 + op2` in Zpx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zpx::sub():

```
void (*sub) (struct calib_Zpx_obj * rop,
const struct calib_Zpx_obj * op1,
const struct calib_Zpx_obj * op2);
```

Set `rop` to `op1 - op2` in Zpx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zpx::neg():

```
void (*neg) (struct calib_Zpx_obj * rop,
const struct calib_Zpx_obj * op);
```

Set `rop` to `- op` in Zpx, where:

`rop` is the Zpx polynomial receiving the result; and
`op` is the Zpx polynomial being negated.

Zpx::mul():

```
void (*mul) (struct calib_Zpx_obj * rop,
const struct calib_Zpx_obj * op1,
const struct calib_Zpx_obj * op2);
```

Set `rop` to `op1 * op2` in Zpx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zpx::mul_z():

```
void (*mul_z) (struct calib_Zpx_obj * rop,
  const struct calib_Zpx_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in Zpx, where:

`rop` is the polynomial receiving the result;
`op1` is the first (polynomial) operand; and
`op2` is the second (GMP integer) operand.

Zpx::ipow():

```
void (*ipow) (struct calib_Zpx_obj * rop,
 const struct calib_Zpx_obj * op,
 int power);
```

Set `rop` to `op ** power` in Zpx, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to exponentiate; and
`power` is the power to take (must be >= 0).

Zpx::dup():

```
struct calib_Zpx_obj *
(*dup) (const struct calib_Zpx_obj * op);
```

Return a dynamically-allocated Zpx polynomial that is a copy of `op`, where:

`op` is the polynomial to be duplicated.

Zpx::free():

```
void (*free) (struct calib_Zpx_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly` is the polynomial to be freed.

This is equivalent to performing `Zpx -> clear (poly);`, followed by `free (poly);`.

Zpx::monicize():

```
void (*monicize)
(struct calib_Zpx_obj * poly);
```

Modify the given Zpx polynomial `poly` to be monic, where: `result`, where:

`poly` is the polynomial to make monic.

Zpx::eval():

```
void (*eval) (mpz_ptr rop,
 const struct calib_Zpx_obj * poly,
 mpz_srcptr value);
```

Evaluate polynomial `poly` at the given `value`, storing the result in `rop`, where:

`rop` is the GMP integer receiving the result;
`poly` is the polynomial to be evaluated; and
`value` is the value at which to evaluate the polynomial.

Zpx::div():

```
void (*div) (struct calib_Zpx_obj * quotient,
struct calib_Zpx_obj * remainder,
const struct calib_Zpx_obj * a,
const struct calib_Zpx_obj * b);
```

Polynomial division in $Zp[x]$, where:

`quotient` receives the quotient polynomial (may be NULL);
`remainder` receives the remainder polynomial (may be NULL);
`a` is the dividend polynomial; and
`b` is the divisor polynomial (may not be zero).

Division in Zp[x] has the following properties:

- $a = quotient * b + remainder$
- $degree(remainder) < degree(b)$

Zpx::gcd():

```
void (*gcd) (struct calib_Zpx_obj * gcd,
const struct calib_Zpx_obj * a,
const struct calib_Zpx_obj * b);
```

Compute the greatest common divisor (GCD) of `a` and `b`, storing the result in `gcd`, where:

`gcd`	receives the resulting GCD polynomial (always monic);
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

Zpx::extgcd():

```
void (*extgcd) (struct calib_Zpx_obj * gcd,
   struct calib_Zpx_obj * xa,
   struct calib_Zpx_obj * xb,
   const struct calib_Zpx_obj * a,
   const struct calib_Zpx_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`gcd`	receives the resulting GCD polynomial (always monic);
`xa`	receives the multiplier polynomial for `a`;
`xb`	receives the multiplier polynomial for `b`;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

The result satisfies the following properties:

1. $degree(xa) < degree(b)$
2. $degree(xb) < degree(a)$

Zpx::factor():

```
struct calib_Zpx_factor *
(*factor) (const struct calib_Zpx_obj * poly);
```

Factor the given polynomial `poly` into its irreducible factors, returning a linked list of these factors, where:

`poly`	is the Zpx polynomial to be factored.

Except for an optional leading constant factor, all other factors are monic and irreducible.

Zpx::factor_square_free():

```
struct calib_Zpx_factor *
(*factor_square_free)
(const struct calib_Zpx_obj * poly);
```

Factor the given polynomial `poly` into square-free factors, where:

`poly`	is the Zpx polynomial to be factored.

Zpx::finish_sqf():

```
struct calib_Zpx_factor *
(*finish_sqf)
(const struct calib_Zpx_factor * sqfactors);
```

Given a list of monic, square-free polynomial `factors`, "finish" the factorization by factoring each such factor into irreducible polynomials, returning a linked-list of these factors, where:

`sqfactors` is a linked-list of primitive, square-free Zpx polynomial factors for which factorization into irreducibles is desired.

Zpx::free_factors():

```
void (*free_factors)
(struct calib_Zpx_factor * factors);
```

Free up the given list of Zpx factors, where:

`factors` is a linked-list factors to be freed.

Zpx::resultant():

```
void (*resultant) (mpz_ptr result,
      const struct calib_Zpx_obj * a,
      const struct calib_Zpx_obj * b);
```

Compute the resultant of Zpx polynomials `a` and `b`, storing the result in `result`, where:

`result` the GMP integer that receives the result;
`a` is the first operand polynomial; and
`b` is the second operand polynomial.

Zpx::derivative():

```
void (*derivative) (struct calib_Zpx_obj * rop,
       const struct calib_Zpx_obj * op);
```

Set `rop` to be the derivative of `op`, where:

`rop` receives the resulting derivative; and
`op` is the polynomial to differentiate.

Zpx::zerop():

```
calib_bool
(*zerop) (const struct calib_Zpx_obj * op);
```

Return 1 if-and-only-if the given Zpx polynomial is identically zero and 0 otherwise, where:

`op` is the Zpx polynomial to test for zero.

Zpx::onep():

```
calib_bool
(*onep) (const struct calib_Zpx_obj * op);
```

Return 1 if-and-only-if the given Zpx polynomial is identically 1 and 0 otherwise, where:

`op` is the Zpx polynomial to test for one.

Zpx::set_genrep():

```
void (*set_genrep) (struct calib_Zpx_obj * rop,
        const struct calib_genrep * op,
        const char * var);
```

Compute a Zpx polynomial obtained from the given genrep `op`, interpreting `var` to be the name of the variable used by the Zpx polynomial, storing the result in `rop`, where:

`rop` receives the resulting Zpx polynomial;
`op` is the genrep to convert into Zpx polynomial form; and
`var` is the variable name (appearing within genrep `op`) that is to be interpreted as the polynomial variable in Zpx.

Zpx::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Zpx_dom * K_of_x,
      const struct calib_Zpx_obj * op,
      const char * var);
```

Return a dynamically-allocated genrep corresponding to the given Zpx polynomial `op`, using `var` as the name of the polynomial variable within the returned genrep, where:

`K_of_x` is the Zpx ring/domain performing this operation;
`op` is the Zpx polynomial to convert into genrep form; and
`var` is the variable name to use in the genrep for the polynomial variable of Zpx.

Zpx::factors_to_genrep():

```
struct calib_genrep *
(*factors_to_genrep)
(const struct calib_Zpx_factor * factors,
 const char * var);
```

Return a dynamically-allocated genrep corresponding to the given list of Zpx `factors`, using `var` as the name of the polynomial variable within the returned genrep, where:

`factors` is the list of Zpx polynomial factors to convert into genrep form; and
`var` is the variable name to use in the genrep for the polynomial variable of Zpx.

Zpx::print_maxima():

```
void (*print_maxima)
(const struct calib_Zpx_obj * op);
```

Print the given Zpx polynomial `op` to stdout using syntax that can be directly read by Maxima, where:

`op` is the Zpx polynomial to be printed.

11 GFpk — The Galois Field $GF(p^k)$

CALIB provides the `GFpk` domain, representing the Galois Field $GF(p^k)$ — the finite field having exactly p^k members, where p is a prime, and $k > 1$ is an integer. The "values" of this domain are represented by the following object:

```
struct calib_GFpk_obj {
const struct calib_GFpk_dom *
dom; /* GF(p^k) domain containing this value */
mpz_ptr coeff; /* Coefficients.  This is an array of */
/* k integers in Z_p.  */
};
```

These value objects are subject to `init()` and `clear()` operations. All such objects must be initialized prior to use by any other CALIB operation. Memory leaks result if they are not cleared when done.

The CALIB `GFpk` domain is constructed by specifying an irreducible "generator" polynomial of degree k in $Z_p[x]$. One may access CALIB's `GFpk` domain as follows:

```
#include "calib/GFpk.h"
...
struct calib_Zpx_obj * gpoly;
struct calib_GFpk_dom * dom;

gpoly = /* generator polynomial in Zp[x] of degree k. */

dom = calib_make_GFpk_dom (gpoly);
...
calib_free_GFpk_dom (dom);
```

There is also a lower-level function to construct a `struct calib_GFpk_dom` object from an array of generator polynomial coefficients:

```
extern struct calib_GFpk_dom *
calib_make_GFpk_dom_z (
const struct calib_Zp_dom * cf,
int k,
mpz_srcptr gcoeff);
```

The `struct calib_GFpk_dom` object contains the following members (pointers to functions) that provide operations of the domain:

GFpk::init():

```
void (*init) (const struct calib_GFpk_dom * f,
 struct calib_GFpk_obj * x);
```

Initialize GFpk value object `x` to be a member of the GFpk domain `f`, where:

`f`	is the GFpk domain performing this operation; and
`x`	is the GFpk value object to be initialized.

GFpk::clear():

```
void (*clear) (struct calib_GFpk_obj * x);
```

Clear out the given GFpk value object `x` (freeing all memory it might hold), where:

`x` is the GFpk value object to be cleared.

GFpk::set():

```
void (*set) (struct calib_GFpk_obj * rop,
const struct calib_GFpk_obj * op);
```

Set `rop` to `op` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result; and
`op` is the source GFpk value object.

GFpk::set_si():

```
void (*set_si) (struct calib_GFpk_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result; and
`op` is the signed integer to convert into $GF(p^k)$ form.

GFpk::set_z():

```
void (*set_z) (struct calib_GFpk_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result; and
`op` is the GMP integer to convert into $GF(p^k)$ form.

GFpk::set_q():

```
void (*set_q) (struct calib_GFpk_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result; and
`op` is the GMP rational to convert into $GF(p^k)$ form.

GFpk::set_var_power():

```
void (*set_var_power)
 (struct calib_GFpk_obj * rop,
  int power);
```

Set `rop` to `a**power` in $GF(p^k)$ (a is the $GF(p^k)$ polynomial variable), where:

`rop` is the GFpk value object receiving the result; and
`power` is the power to set (must be non-negative).

GFpk::add():

```
void (*add) (struct calib_GFpk_obj * rop,
const struct calib_GFpk_obj * op1,
```

```
const struct calib_GFpk_obj * op2);
```

Set `rop` to `op1 + op2` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

GFpk::sub():

```
void (*sub) (struct calib_GFpk_obj * rop,
const struct calib_GFpk_obj * op1,
const struct calib_GFpk_obj * op2);
```

Set `rop` to `op1 - op2` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

GFpk::neg():

```
void (*neg) (struct calib_GFpk_obj * rop,
mpz_srcptr op);
```

Set `rop` to `- op` in `GF(p^k)`, where:

`rop` is the GFpk value object receiving the result; and
`op` is the operand being negated.

GFpk::mul():

```
void (*mul) (struct calib_GFpk_obj * rop,
const struct calib_GFpk_obj * op1,
const struct calib_GFpk_obj * op2);
```

Set `rop` to `op1 * op2` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

GFpk::mul_z():

```
void (*mul_z) (struct calib_GFpk_obj * rop,
  const struct calib_GFpk_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op1` is the first operand; and
`op2` is the second operand (a single GMP integer).

GFpk::mul_a():

```
void (*mul_a) (struct calib_GFpk_obj * rop,
```

```
    const struct calib_GFpk_obj * op);
```

Set `rop` to `op * a` (mulitply by the generator polynomial variable 'a') in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op` is the operand being multiplied.

GFpk::ipow():

```
void (*ipow) (struct calib_GFpk_obj * rop,
 const struct calib_GFpk_obj * op,
 int power);
```

Set `rop` to `op ** power` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op` is the operand to exponentiate; and
`power` is the power to take (may be positive, negative or zero).

GFpk::inv():

```
void (*inv) (struct calib_GFpk_obj * rop,
const struct calib_GFpk_obj * op);
```

Set `rop` to the multiplicative inverse of `op` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result; and
`op` is the operand to invert (must not be zero).

GFpk::pth_root():

```
void (*pth_root) (struct calib_GFpk_obj * rop,
     const struct calib_GFpk_obj * op);
```

Set `rop` to the p-th root of `op` in $GF(p^k)$, where:

`rop` is the GFpk value object receiving the result;
`op` is the operand whose p-th root is computed.

The p-th root of *op* is the element y in $GF(p^k)$ such that $y^p = op$. This can be calculated as $y = op^M$, where $M = p^(k-1)$.

GFpk::cvZa():

```
void (*cvZa) (struct calib_GFpk_obj * rop,
 const struct calib_Za_obj * op);
```

Convert the given Za value `op` into the corresponding element of target $GF(p^k)$ domain, storing the result in `rop`, where:

`rop` is the GFpk value object receiving the result; and
`op` is the element of $Z(a)$ (array of GMP integers) to convert.

It is intended that the generator polynomial defining $Z(a)$ may be different from the generator polynomial defining $GF(p^k)$.

GFpk::degree():

```
int (*degree) (const struct calib_GFpk_obj * op);
```

Return the degree (in variable a of the $GF(p^k)$ generator polyomial) of the given element `op` of $GF(p^k)$, where:

`op` is the GFpk value object whose degree is to be returned.

Note that the degree of a zero value is -1.

GFpk::zerop():

```
calib_bool
(*zerop) (const struct calib_GFpk_obj * op);
```

Return 1 if-and-only-if the given GFpk value object is identically zero and 0 otherwise, where:

`op` is the GFpk value object to test for zero.

GFpk::onep():

```
calib_bool
(*onep) (const struct calib_GFpk_obj * op);
```

Return 1 if-and-only-if the given GFpk value object is identically 1 and 0 otherwise, where:

`op` is the GFpk value object to test for one.

GFpk::set_random():

```
void (*set_random) (struct calib_GFpk_obj * rop,
         struct calib_Random * randp);
```

Set `rop` to be a random element of $GF(p^k)$ using the given random number generator, where:

`rop` is the GFpk value to set to a randomized value; and
`randp` is the random number generator to use.

GFpk::set_genrep():

```
void (*set_genrep) (struct calib_GFpk_obj * rop,
         struct calib_genrep * op,
         const char * var);
```

Given a genrep `op` and the name `var` of the $GF(p^k)$ generator polynomial variable (as it appears in `op`), convert this genrep into a value in this $GF(p^k)$ domain, storing the result in `rop`, where:

`rop` is the GFpk value receiving the result;
`op` is the genrep being converted; and
`var` is the name of the $GF(p^k)$ generator polyomial variable appearing within genrep `op`.

GFpk::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_GFpk_obj * op,
```

```
const char * var);
```

Return a dynamically-allocated genrep representing the given GFpk value `op`, using the given variable name `var` to represent the variable of the $GF(p^k)$ generator polynomial, where:

`op` is the GFpk value object to convert; and

`var` is the name to use when representing the variable of the $GF(p^k)$ generator polyomial within the genrep returned.

12 GFpkx — The Polynomial Ring $GF(p^k)[x]$

CALIB provides the `GFpkx` domain, representing the ring $GF(p^k)[x]$, the univariate polynomials having coefficients that are in the Galois Field $GF(p^k)$. The "values" of this domain are represented by the following object:

```
struct calib_GFpkx_obj {
int degree; /* Degree of polynomial */
int size; /* Size of coeff buffer (degree < size) */
const struct calib_GFpkx_dom *
dom; /* Domain of polynomial */
struct calib_GFpk_obj *
coeff; /* Coefficients of polynomial.          */
};
```

The following additional object is used to represent linked-lists of factors produced by various GFpkx factorization algorithms:

```
struct calib_GFpkx_factor {
int multiplicity;
struct calib_GFpkx_obj * factor;
struct calib_GFpkx_factor * next;
};
```

CALIB `GFpkx` domains are constructed by specifying a `GFpk` domain used to represent the coefficients of the corresponding GFpkx polynomials.

One may access CALIB's `GFpkx` domain as follows:

```
#include "calib/GFpkx.h"
...
struct calib_Zpx_obj * gpoly;
struct calib_GFpk_dom * GFpk;
        struct calib_GFpkx_dom * GFpkx;
struct calib_GFpkx_obj poly1, poly2;
...
gpoly = /* Make generator polynomial in Zp[x]. */
GFpk  = calib_make_GFpk_dom (gpoly);
GFpkx = calib_make_GFpkx_dom (GFpk);
...
GFpkx -> init (&poly1);
GFpkx -> init (&poly2);
...
GFpkx -> mul (GFpkx, &poly1, &poly1, &poly2);
...
GFpkx -> clear (&poly2);
GFpkx -> clear (&poly1);
calib_free_GFpkx_dom (GFpkx);
calib_free_GFPk_dom (GFpk);
...
```

The `struct calib_GFpkx_dom` object contains the following members (pointers to functions) that provide operations of the domain:

GFpkx::init():

```
void (*init) (const struct calib_GFpkx_dom * K_of_x,
 struct calib_GFpkx_obj * x);
```

Initialize the given GFpkx polynomial `x`, where:

`K_of_x`	is the GFpkx ring/domain the polynomial belongs to; and
`x`	is the polynomial to initialize.

GFpkx::init_degree():

```
void (*init_degree)
 (const struct calib_GFpkx_dom * K_of_x,
  struct calib_GFpkx_obj * x,
  int degree);
```

Initialize the given GFpkx polynomial `x` (while assuring that internal buffers are sufficiently large to hold a polynomial of up to the given `degree` without further allocation), where:

`K_of_x`	is the GFpkx ring/domain the polynomial belongs to;
`x`	is the polynomial to initialize; and
`degree`	is the guaranteed minimimum degree polynomial that `x` will be able to hold (without further buffer allocation) upon successful completion of this operation.

GFpkx::alloc():

```
void (*alloc) (struct calib_GFpkx_obj * rop,
  int degree);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial of at least the given `degree`, where:

`rop`	is the polynomial whose allocation is to be adjusted; and
`degree`	is the guaranteed minimum degree polynomial that `rop` will be able to hold (without further buffer allocation) upon successful completion of this operation.

GFpkx::clear():

```
void (*clear) (struct calib_GFpkx_obj * x);
```

Clear out the given polynomial `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x` is the polynomial to be cleared.

GFpkx::set():

```
void (*set) (struct calib_GFpkx_obj * rop,
const struct calib_GFpkx_obj * op);
```

Set `rop` to `op` in GFpkx, where:

`rop`	is the polynomial receiving the result;
`rop`	is the polynomial to copy.

GFpkx::set_si():

```
void (*set_si) (struct calib_GFpkx_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in GFpkx, where:

`rop` is the polynomial receiving the result;
`op` is the integer value to set.

GFpkx::set_z():

```
void (*set_z) (struct calib_GFpkx_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in GFpkx, where:

`rop` is the polynomial receiving the result;
`op` is the GMP integer value to set.

GFpkx::set_q():

```
void (*set_q) (struct calib_GFpkx_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in GFpkx, where:

`rop` is the polynomial receiving the result;
`op` is the GMP rational value to set.

GFpkx::set_var_power():

```
void (*set_var_power)
 (struct calib_GFpkx_obj * rop,
  int power);
```

Set `rop` to `x ** power` in GFpkx, where:

`rop` is the Zpx polynomial receiving the result;
`power` is the power to set (must be non-negative).

GFpkx::add():

```
void (*add) (struct calib_GFpkx_obj * rop,
const struct calib_GFpkx_obj * op1,
const struct calib_GFpkx_obj * op2);
```

Set `rop` to `op1 + op2` in GFpkx, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

GFpkx::sub():

```
void (*sub) (struct calib_GFpkx_obj * rop,
const struct calib_GFpkx_obj * op1,
const struct calib_GFpkx_obj * op2);
```

Set `rop` to `op1 - op2` in GFpkx, where:

`rop`	is the polynomial receiving the result;
`op1`	is the first operand; and
`op2`	is the second operand.

GFpkx::neg():

```
void (*neg) (struct calib_GFpkx_obj * rop,
const struct calib_GFpkx_obj * op);
```

Set `rop` to `- op` in GFpkx, where:

`rop`	is the GFpkx polynomial receiving the result; and
`op`	is the GFpkx polynomial being negated.

GFpkx::mul():

```
void (*mul) (struct calib_GFpkx_obj * rop,
const struct calib_GFpkx_obj * op1,
const struct calib_GFpkx_obj * op2);
```

Set `rop` to `op1 * op2` in GFpkx, where:

`rop`	is the polynomial receiving the result;
`op1`	is the first operand; and
`op2`	is the second operand.

GFpkx::mul_z():

```
void (*mul_z) (struct calib_GFpkx_obj * rop,
  const struct calib_GFpkx_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in GFpkx, where:

`rop`	is the polynomial receiving the result;
`op1`	is the first (polynomial) operand; and
`op2`	is the second (GMP integer) operand.

GFpkx::ipow():

```
void (*ipow) (struct calib_GFpkx_obj * rop,
 const struct calib_GFpkx_obj * op,
 int power);
```

Set `rop` to `op ** power` in GFpkx, where:

`rop`	is the polynomial receiving the result;
`op`	is the polynomial to exponentiate; and
`power`	is the power to take (must be >= 0).

GFpkx::dup():

```
struct calib_GFpkx_obj *
(*dup) (const struct calib_GFpkx_obj * op);
```

Return a dynamically-allocated GFpkx polynomial that is a copy of `op`, where:

`op` is the polynomial to be duplicated.

GFpkx::free():

```
void (*free) (struct calib_GFpkx_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly` is the polynomial to be freed.

This is equivalent to performing `GFpkx -> clear (poly);`, followed by `free (poly);`.

GFpkx::eval():

```
void (*eval) (struct calib_GFpk_obj * rop,
 const struct calib_GFpkx_obj * poly,
 const struct calib_GFpk_obj * value);
```

Evaluate polynomial `poly` at the given `value`, storing the result in `rop`, where:

`rop` is the GFpk value object receiving the result;
`op` is the polynomial to be evaluated; and
`value` is the GFpk value at which to evaluate the polynomial.

GFpkx::div():

```
void (*div) (struct calib_GFpkx_obj * quotient,
struct calib_GFpkx_obj * remainder,
const struct calib_GFpkx_obj * a,
const struct calib_GFpkx_obj * b);
```

Polynomial division in $GF(p^k)[x]$, where:

`quotient` receives the quotient polynomial (may be NULL);
`remainder` receives the remainder polynomial (may be NULL);
`a` is the dividend polynomial; and
`b` is the divisor polynomial (may not be zero).

Division in $GF(p^k)[x]$ has the following properties:

- $a = quotient * b + remainder$
- $degree(remainder) < degree(b)$

GFpkx::gcd():

```
void (*gcd) (struct calib_GFpkx_obj * gcd,
const struct calib_GFpkx_obj * a,
const struct calib_GFpkx_obj * b);
```

Compute the greatest common divisor (GCD) of `a` and `b`, storing the result in `gcd`, where:

`gcd` receives the resulting GCD polynomial (always monic);
`a` is the first operand polynomial; and
`b` is the second operand polynomial.

The GCD is always monic unless a = b = 0.

GFpkx::extgcd():

```
void (*extgcd) (struct calib_GFpkx_obj * gcd,
   struct calib_GFpkx_obj * xa,
   struct calib_GFpkx_obj * xb,
   const struct calib_GFpkx_obj * a,
   const struct calib_GFpkx_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`K_of_x`	is the GFpkx ring/domain performing this operation;
`gcd`	receives the resulting GCD polynomial (always monic);
`xa`	receives the multiplier polynomial for `a`;
`xb`	receives the multiplier polynomial for `b`;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

The result satisfies the following properties:

1. $degree(xa) < degree(b)$
2. $degree(xb) < degree(a)$

GFpkx::cvZax():

```
struct calib_GFpkx_obj *
(*cvZax) (const struct calib_GFpkx_dom * K_of_x,
  const struct calib_Zax_obj * op);
```

Return a dynamically-allocated GFpkx polynomial that is copied from the given Zax polynomial `op`, where:

`K_of_x`	is the destination GFpkx ring/domain;
`poly`	is the Zax polynomial to be copied into GFpkx form.

Note: this function requires that `K_of_x` and the Zax domain of `op` have the same generator polynomial — it is a *fatal* error if they do not.

GFpkx::factor():

```
struct calib_GFpkx_factor *
(*factor) (const struct calib_GFpkx_obj * poly);
```

Factor the given polynomial `poly` into its irreducible factors, returning a linked list of these factors, where:

`poly`	is the GFpkx polynomial to be factored.

Except for an optional leading constant factor, all other factors are monic and irreducible.

GFpkx::free_factors():

```
void (*free_factors)
(struct calib_GFpkx_factor * factors);
```

Free up the given list of GFpkx factors, where:

`factors` is a linked-list factors to be freed.

GFpkx::set_random():

```
void (*set_random) (struct calib_GFpkx_obj * rop,
        int d,
        struct calib_Random * randp);
```

Set `rop` to be a randomly chosen GFpkx polynomial of degree `d`, using random numbers from `randp`, where:

`rop` receives the GFpkx polynomial result;
`d` is the degree of polynomial to generate; and
`randp` is the random number generator to use.

GFpkx::zerop():

```
calib_bool
(*zerop) (const struct calib_GFpkx_obj * op);
```

Return 1 if-and-only-if the given GFpkx polynomial is identically zero and 0 otherwise, where:

`op` is the GFpkx polynomial to test for zero.

GFpkx::onep():

```
calib_bool
(*onep) (const struct calib_GFpkx_obj * op);
```

Return 1 if-and-only-if the given GFpkx polynomial is identically one and 0 otherwise, where:

`op` is the GFpkx polynomial to test for one.

GFpkx::set_genrep():

```
void (*set_genrep) (struct calib_GFpkx_obj * rop,
        const struct calib_genrep * op,
        const char * xvar,
        const char * avar);
```

Compute a GFpkx polynomial obtained from the given genrep `op`, interpreting `xvar` to be the name of the variable used by the GFpkx polynomial and `avar` to be the name of the variable used by the GFpk coefficients, storing the result in `rop`, where:

`rop` receives the GFpkx polynomial result;
`op` is the genrep to convert into GFpkx polynomial form;
`xvar` is the variable name (appearing within genrep `op`) that is to be interpreted as the polynomial variable in GFpkx; and
`avar` is the variable name (appearing within genrep `op`) that is to be interpreted as the variable used by the GFpk coefficients.

GFpkx::to_genrep():

```
struct calib_genrep *
```

```
(*to_genrep) (const struct calib_GFpkx_obj * op,
      const char * xvar,
      const char * avar);
```

Return a dynamically-allocated genrep corresponding to the given GFpkx polynomial `op`, using `xvar` as the name of the GFpkx polynomial variable and `avar` as the name of the GFpk coefficient variable within the returned genrep, where:

`op` is the GFpkx polynomial to convert into genrep form;

`xvar` is the variable name to use in the genrep for the polynomial variable of GFpkx; and

`avar` is the variable name to use in the genrep for the GFpk coefficients.

GFpkx::factors_to_genrep():

```
struct calib_genrep *
(*factors_to_genrep)
(const struct calib_GFpkx_factor * factors,
 const char * xvar,
 const char * avar);
```

Return a dynamically-allocated genrep corresponding to the given list of GFpkx `factors`, using `xvar` as the name of the GFpkx polynomial variable and `avar` as the name of the GFpk coefficient variable within the returned genrep, where:

`factors` is the list of GFpkx polynomial factors to convert into genrep form; and

`xvar` is the variable name to use in the genrep for the polynomial variable of GFpkx;

`avar` is the variable name to use in the genrep for the GFpk coefficients.

13 Za — The Ring $Z(a)$

CALIB provides the `Za` domain, representing the ring $Z(a)$, the extension of the integers with given algebraic integer a (specified via a monic irreducible polynomial in $Z[x]$ of degree at least two). The "values" of this domain are represented by the following object:

```
struct calib_Za_obj {
const struct calib_Za_dom *
dom; /* Z(a) domain containing this value */
mpz_ptr coeff; /* Coefficients.  This is an array of */
/* k integers.  */
};
```

These value objects are subject to `init()` and `clear()` operations. All such objects must be initialized prior to use by any other CALIB operation. Memory leaks result if they are not cleared when done.

The CALIB `Za` domain is constructed by specifying a monic, irreducible "generator" polynomial of degree k in $Z[x]$. One may access CALIB's `Za` domain as follows:

```
#include "calib/Za.h"
...
struct calib_Zx_obj * apoly;
struct calib_Za_dom * Za;

apoly = /* monic, irreducible polynomial defining 'a'. */

Za = calib_make_Za_dom (apoly);
...
calib_free_Za_dom (Za);
```

Let k be the degree of the monic polynomial defining the algebraic integer a. The "values" of this Za domain are polynomials in $Z[a]$ having degree at most $k - 1$.

The `struct calib_Za_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Za::init():

```
void (*init) (const struct calib_Za_dom * rp,
 struct calib_Z_obj * x);
```

Initialize Za value object `x` to be a member of the Za domain `rp`, where:

`rp` is the Za ring/domain performing this operation; and
`x` is the Za value object to be initialized.

Za::clear():

```
void (*clear) (struct calib_Z_obj * x);
```

Clear out the given Za value object `x` (freeing all memory it might hold), where:

`x` is the Za value object to be cleared.

Za::set():

```
void (*set) (struct calib_Za_obj * rop,
```

```
const struct calib_Za_obj * op);
```

Set rop to op in Za, where:

rop — is the destination Z(a) value; and
op — is the source Z(a) value.

Za::set_si():

```
void (*set_si) (struct calib_Za_obj * rop,
   calib_si_t op);
```

Set rop to op in Za, where:

rop — is the destination Z(a) value; and
op — is the source integer value.

Za::set_z():

```
void (*set_z) (struct calib_Za_obj * rop,
  mpz_srcptr op);
```

Set rop to op in Za, where:

rop — is the destination Z(a) value; and
op — is the source GMP integer value.

Za::set_q():

```
void (*set_q) (struct calib_Za_obj * rop,
  mpq_srcptr op);
```

Set rop to op in Za, where:

rp — is the Za ring/domain performing this operation;
rop — is the destination Z(a) value; and
op — is the source GMP integer value.

The denominator of op *must* be 1.

Za::set_var_power():

```
void (*set_var_power)
 (struct calib_Za_obj * rop,
  int power);
```

Set rop to a ** power in Za, where:

rop — is the destination Z(a) value; and
power — is the power to set (must be non-negative).

Za::add():

```
void (*add) (struct calib_Za_obj * rop,
const struct calib_Za_obj * op1,
const struct calib_Za_obj * op2);
```

Set rop to op1 + op2 in Za, where:

`rop` is the destination Z(a) value;
`a` is the first operand; and
`b` is the second operand.

Za::sub():

```
void (*sub) (struct calib_Za_obj * rop,
const struct calib_Za_obj * op1,
const struct calib_Za_obj * op2);
```

Set `rop` to `op1 - op2` in Za, where:

`rop` is the destination Z(a) value;
`op1` is the first operand; and
`op2` is the second operand.

Za::neg():

```
void (*neg) (struct calib_Za_obj * rop,
const struct calib_Za_obj * op);
```

Set `rop` to `- op` in Za, where:

`rop` is the destination Z(a) value; and
`op` is the operand to be negated.

Za::mul():

```
void (*mul) (struct calib_Za_obj * rop,
const struct calib_Za_obj * op1,
const struct calib_Za_obj * op2);
```

Set `rop` to `op1 * op2` in Za, where:

`rop` is the destination Z(a) value;
`op1` is the first operand; and
`op2` is the second operand.

Za::mul_z():

```
void (*mul_z) (struct calib_Za_obj * rop,
  const struct calib_Za_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in Za, where:

`rop` is the destination Z(a) value;
`op1` is the first operand; and
`op2` is the second operand.

Za::mul_a():

```
void (*mul_a) (struct calib_Za_obj * rop,
  const struct calib_Za_obj * op);
```

Set `rop` to `op * a` (multiply by algebraic integer a), where:

`rop` is the destination Z(a) value; and
`op` is the source operand being multiplied.

Za::ipow():

```
void (*ipow) (struct calib_Za_obj * rop,
 const struct calib_Za_obj * op,
 int power);
```

Set `rop` to `op ** power` in Za, where:

`rop` is the destination Z(a) value;
`op` is the operand to exponentiate; and
`power` is the power to take (must be >= 0).

Za::pinv():

```
void (*pinv) (struct calib_Za_obj * rop,
 mpz_ptr d,
 const struct calib_Za_obj * op);
```

Pseudo-inverse in Z(a). Compute *rop* in $Z(a)$ and *d* in Z such that $rop * op = d$, where:

`rop` is the destination Z(a) value;
`d` is a single GMP integer receiving the value *d*; and
`op` is the source operand to pseudo-invert.

Za::div_z_exact():

```
void (*div_z_exact)
(struct calib_Za_obj * rop,
 const struct calib_Za_obj * op1,
 mpz_srcptr op2);
```

Set `rop` to `op1 / op2` in Za, where:

`rop` is the destination Z(a) value;
`op1` is the first operand; and
`op2` is the second operand.

The division must be exact.

Za::prim_part():

```
void (*prim_part) (mpz_ptr content,
      struct calib_Za_obj * ppart,
      const struct calib_Za_obj * op);
```

Compute content and primitive part of `op` in Z(a), storing them in `content` and `ppart`, respectively, where:

`content` receives the content of `op`;
`ppart` is the Z(a) value object receiving the primitive part of `op`; and
`op` is the operand for which to compute the content and primitive part.

Za::cvZa():

```
void (*cvZa) (struct calib_Za_obj * rop,
```

```
const struct calib_Za_obj * op);
```

Set rop to op. Note that rop and op are permitted to belong to different Za domains, and this routine performs the necessary conversion, where:

rop is the destination Z(a) value; and
op is the source Z(a) value to convert.

Za::degree():

```
int (*degree) (const struct calib_Za_obj * op);
```

Return the degree (in a) of the given $Z(a)$ value op, where:

op is the operand for which to get the degree.

Note that the degree of a zero value is -1.

Za::zerop():

```
calib_bool
(*zerop) (const struct calib_Za_obj * op);
```

Return 1 if-and-only-if op is identically zero and 0 otherwise, where:

op is the operand to test for zero.

Za::onep():

```
calib_bool
(*onep) (const struct calib_Za_obj * op);
```

Return 1 if-and-only-if op is identically one and 0 otherwise, where:

op is the operand to test for one.

Za::set_genrep():

```
void (*set_genrep) (struct calib_Za_obj * rop,
        const struct calib_genrep * op,
        const char * var);
```

Given a genrep op and the name var of the algebraic integer a, convert this genrep into a value in this $Z(a)$ domain, storing the result in rop, where:

rop is the Z(a) value object receiving value;
op is the genrep being converted; and
var is the name of the algebraic integer a appearing within genrep op.

Za::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Za_obj * op,
      const char * var);
```

Return a dynamically-allocated genrep representing the given value op of the given $Z(a)$ domain, using the given variable name var to represent the algebraic integer a, where:

op is value from $Z(a)$ being converted to genrep form; and `var` is the name of the algebraic integer `a` as it should appear within the genrep returned.

14 Zax — The Polynomial Ring $Z(a)[x]$

CALIB provides the `Zax` domain, representing the ring $Z(a)[x]$, the univariate polynomials having coefficients that are an algebraic extension of the integers. The "values" of this domain are represented by the following object:

```
struct calib_Zax_obj {
int degree; /* Degree of polynomial */
int size; /* Size of coeff buffer (degree < size) */
const struct calib_Zax_dom *
dom; /* Domain of polynomial */
mpz_ptr coeff; /* Coefficients of polynomial.  */
};
```

The CALIB `Zax` domain is constructed by specifying a `Za` domain used to represent the coefficients of the corresponding Zax polynomials.

One may access CALIB's `Zax` domain as follows:

```
#include "calib/Zax.h"
...
struct calib_Zx_obj * gpoly;
struct calib_Za_dom * Za;
        struct calib_Zax_dom * Zax;
struct calib_Zax_obj poly1, poly2;
...
gpoly = /* generator poly in Z[x] */
Za = calib_make_Za_dom (gpoly);
Zax = calib_make_Zax_dom (Za);
...
Zax -> init (&poly1);
Zax -> init (&poly2);
...
Zax -> mul (&poly1, &poly1, &poly2);
...
Zax -> clear (&poly2);
Zax -> clear (&poly1);
calib_free_Zax_dom (Zax);
calib_free_Za_dom (Za);
```

The `struct calib_Zax_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Zax::init():

```
void (*init) (const struct calib_Zax_dom * R_of_x,
 struct calib_Zax_obj * x);
```

Initialize the given Zax polynomial `x`, where:

`R_of_x`	is the Zax ring/domain the polynomial belongs to; and
`x`	is the polynomial to initialize.

Zax::init_degree():

```
void (*init_degree)
(const struct calib_Zax_dom * R_of_x,
 struct calib_Zax_obj * x,
 int degree);
```

Initialize the given Zax polynomial `x` (while assuring that internal buffers are sufficiently large to hold a polynomial of up to the given `degree` without further allocation), where:

`R_of_x`	is the Zax ring/domain the polynomial belongs to;
`x`	is the polynomial to initialize; and
`degree`	is the guaranteed minimimum degree polynomial that `x` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Zax::init_si():

```
void (*init_si)
(const struct calib_Zax_dom * R_of_x,
                                 struct calib_Zax_obj * x,
 calib_si_t op);
```

Initialize the given Zax polynomial `x` to have the constant value `op` (while assuring that internal buffers are sufficiently large to hold a polynomial of up to the given `degree` without further allocation), where:

`R_of_x`	is the Zax ring/domain performing this operation;
`x`	is the polynomial to initialize; and
`op`	is the constant value to which polynomial `x` is set.

Zax::alloc():

```
void (*alloc) (struct calib_Zax_obj * rop,
  int degree);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial of at least the given `degree`, where:

`rop`	is the polynomial whose allocation is to be adjusted; and
`degree`	is the guaranteed minimum degree polynomial that `rop` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Zax::clear():

```
void (*clear) (struct calib_Zax_obj * x);
```

Clear out the given polynomial `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x`	is the polynomial to be cleared.

Zax::set():

```
void (*set) (struct calib_Zax_obj * rop,
```

```
const struct calib_Zax_obj * op);
```

Set rop to op in Zax, where:

rop — is the polynomial receiving the result;
op — is the polynomial to copy.

Zax::set_si():

```
void (*set_si) (struct calib_Zax_obj * rop,
   calib_si_t op);
```

Set rop to op in Zax, where:

rop — is the polynomial receiving the result;
op — is the integer value to set.

Zax::set_z():

```
void (*set_z) (struct calib_Zax_obj * rop,
  mpz_srcptr op);
```

Set rop to op in Zax, where:

rop — is the polynomial receiving the result;
op — is the GMP integer value to set.

Zax::set_q():

```
void (*set_q) (struct calib_Zax_obj * rop,
  mpq_srcptr op);
```

Set rop to op in Zax, where:

rop — is the polynomial receiving the result;
op — is the GMP rational value to set (must have denominator of 1).

Zax::set_var_power():

```
void (*set_var_power)
 (struct calib_Zax_obj * rop,
  int power);
```

Set rop to x ** power in Zax, where:

rop — is the Zpx polynomial receiving the result;
power — is the power to set (must be non-negative).

Zax::add():

```
void (*add) (struct calib_Zax_obj * rop,
const struct calib_Zax_obj * op1,
const struct calib_Zax_obj * op2);
```

Set rop to op1 + op2 in Zax, where:

rop — is the polynomial receiving the result;
op1 — is the first operand; and

`op2` is the second operand.

Zax::sub():

```
void (*sub) (struct calib_Zax_obj * rop,
const struct calib_Zax_obj * op1,
const struct calib_Zax_obj * op2);
```

Set `rop` to `op1 - op2` in Zax, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zax::neg():

```
void (*neg) (struct calib_Zax_obj * rop,
const struct calib_Zax_obj * op);
```

Set `rop` to `- op` in Zax, where:

`rop` is the Zax polynomial receiving the result; and
`op` is the Zax polynomial being negated.

Zax::mul():

```
void (*mul) (struct calib_Zax_obj * rop,
const struct calib_Zax_obj * op1,
const struct calib_Zax_obj * op2);
```

Set `rop` to `op1 * op2` in Zax, where: `result`, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Zax::mul_z():

```
void (*mul_z) (struct calib_Zax_obj * rop,
  const struct calib_Zax_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in Zax, where:

`rop` is the polynomial receiving the result;
`op1` is the first (polynomial) operand; and
`op2` is the second (GMP integer) operand.

Zax::ipow():

```
void (*ipow) (struct calib_Zax_obj * rop,
 const calib_Zax_obj * op,
 int power);
struct calib_Zax_obj * poly);
```

Set `rop` to `op ** power` in Zax, where:

`rop`	is the polynomial receiving the result;
`op`	is the polynomial to exponentiate; and
`power`	is the power to take (must be >= 0).

Zax::dup():

```
struct calib_Zax_obj *
(*dup) (const struct calib_Zax_obj * op);
```

Return a dynamically-allocated Zax polynomial that is a copy of `op`, where:

`op`	is the polynomial to be duplicated.

Zax::free():

```
void (*free) (struct calib_Zax_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly`	is the polynomial to be freed.

This is equivalent to performing `Zax -> clear (poly);`, followed by `free (poly);`.

Zax::eval():

```
void (*eval) (struct calib_Za_obj * rop,
 const struct calib_Zax_obj * op,
 const struct calib_Za_obj * value);
```

Evaluate polynomial `op` at the given `value`, storing the result in `rop`, where:

`rop`	is the Za value object receiving the result;
`op`	is the polynomial to be evaluated; and
`value`	is the Za value at which to evaluate `op`.

Zax::div():

```
void (*div) (struct calib_Zax_obj * quotient,
struct calib_Zax_obj * remainder,
mpz_ptr d,
const struct calib_Zax_obj * a,
const struct calib_Zax_obj * b);
```

Polynomial division in $Zp[x]$, where:

`quotient`	receives the quotient polynomial (may be NULL);
`remainder`	receives the remainder polynomial (may be NULL);
`d`	receives the common denominator (may be NULL);
`a`	is the dividend polynomial; and
`b`	is the divisor polynomial (may not be zero).

Division in Z(a)[x] has the following properties:

- $d * a = quotient * b + remainder$
- $degree(remainder) < degree(b)$

Zax::div_z_exact():

```
void (*div_z_exact)
```

```
(struct calib_Zax_obj * rop,
 const struct calib_Zax_obj * op1,
 mpz_srcptr op2);
```

Set `rop` to `op1 / op2` in Zax, where:

`rop`	is the polynomial receiving the result;
`op1`	is the first (polynomial) operand; and
`op2`	is the second (GMP integer) operand.

The division must be exact or a *fatal* error results.

Zax::extgcd():

```
void (*extgcd) (struct calib_Zax_obj * gcd,
   struct calib_Zax_obj * xa,
   struct calib_Zax_obj * xb,
   const struct calib_Zax_obj * a,
   const struct calib_Zax_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`gcd`	receives the resulting GCD polynomial (monic unless a = b = 0);
`xa`	receives the multiplier polynomial for `a`;
`xb`	receives the multiplier polynomial for `b`;
`a`	is the first operand polynomial; and
`b`	is the second operand polynomial.

The result satisfies the following properties:

1. $degree(xa) < degree(b)$
2. $degree(xb) < degree(a)$

Zax::cvZax():

```
struct calib_Zax_obj *
(*cvZax) (const struct calib_Zax_dom * Zax
  const struct calib_Zax_obj * op);
```

Return a dynamically-allocated Zax polynomial that is copied from the given Zax polynomial `op`, where:

`Zax`	is the source Zax ring/domain;
`op`	is the Zax polynomial to be copied into Zax form.

The generator polynomials defining `Zax` and the domain of `op` are intended to be different.

Zax::cvGFpkx():

```
struct calib_Zax_obj *
(*cvGFpkx) (const struct calib_Zax_dom * K_of_x,
  const struct calib_GFpkx_dom * gfpkx,
  const struct calib_GFpkx_obj * gfpoly);
```

Return a dynamically-allocated Zax polynomial that is copied from the given GFpkx polynomial `src`, where:

`K_of_x` is the Zax ring/domain performing this operation;
`gfpkx` is the GFpkx ring/domain for `gfpoly`; and
`gfpoly` is the GFpkx polynomial to be copied into Zax form.

The generator polynomials defining `K_of_x` and `gfpkx` must be identical or a *fatal* error occurs.

Zax::cvQax():

```
struct calib_Zax_obj *
(*cvQax) (const struct calib_Zax_dom * R_of_x,
  const struct calib_Qax_obj * qpoly,
  mpz_srcptr d);
```

Return a dynamically-allocated Zax polynomial that is copied from the given Qax polynomial `qpoly` (after multiplying by `d` to clear all denominators), where:

`R_of_x` is the Zax ring/domain performing this operation;
`qpoly` is the source Qax polynomial to convert; and
`d` is the common demoninator by which to multiply.

The generator polynomials defining `R_of_x` and the domain of `qpoly` must be identical or a *fatal* error occurs. It is also a *fatal* error if any non-integral coefficients remain after multiplying by `d`.

Zax::zerop():

```
calib_bool
(*zerop) (const struct calib_Zax_obj * op);
```

Return 1 if-and-only-if the given Zax polynomial is identically zero and 0 otherwise, where:

`op` is the Zax polynomial to test for zero.

Zax::onep():

```
calib_bool
(*onep) (const struct calib_Zax_obj * op);
```

Return 1 if-and-only-if the given Zax polynomial is identically one and 0 otherwise, where:

`op` is the Zax polynomial to test for one.

Zax::set_genrep():

```
void (*set_genrep) (struct calib_Zax_obj * rop,
       const struct calib_genrep * op,
       const char * xvar,
       const char * avar);
```

Comput a Zax polynomial obtained from the given genrep `op`, interpreting `xvar` to be the name of the variable used by the Zax polynomial and `avar` to be the name of the variable denoting the algebraic number, storing the result in `rop`, where:

`rop` receives the resulting Zax polynomial;

`op` is the genrep to convert into Zax polynomial form;

`xvar` is the variable name (appearing within genrep `op`) that is interpreted as the polynomial variable in Zax; and

`avar` is the variable name (appearing within genrep `op`) that is interpreted as being the algebraic number.

Zax::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Zax_obj * op,
      const char * xvar,
      const char * avar);
```

Return a dynamically-allocated genrep corresponding to the given Zax polynomial `op`, using `xvar` as the name of the polynomial variable within the returned genrep and `avar` as the name of the algebraic number, where:

`op` is the Zax polynomial to convert into genrep form;

`xvar` is the variable name to use in the genrep for the polynomial variable of Zax; and

`avar` is the variable name to use in the genrep for the algebraid number.

15 Qa — The Field $Q(a)$

CALIB provides the `Qa` domain, representing the field $Q(a)$, the extension of the rationals with given algebraic number a (specified via an irreducible polynomial in $Z[x]$ of degree at least 2). The "values" of this domain are represented by the following object:

```
/*
 * An instance of a value in Q(a).  We represent this as the product
 * of a rational multiplier times a primitive Z[x] polynomial whose leading
 * coefficient (if any) is strictly positive.
 */

struct calib_Qa_obj {
mpq_t qfact; /* Rational multiplier */
const struct calib_Qa_dom *
dom; /* Q(a) domain containing this value */
int degree; /* Degree of this value in a */
mpz_ptr coeff; /* Coefficients.  This is always an */
/* array of k integers so that */
/* reallocation is never required */
};
```

These objects are subject to `init()` and `clear()` operations. Memory leaks result if they are not cleared when done.

The CALIB `Qa` domain is constructed by specifying an irreducible polynomial in $Z[x]$ of degree at least 2.

One may access CALIB's `Qa` domain as follows:

```
#include "calib/Qa.h"
...
struct calib_Zx_obj * apoly;
struct calib_Qa_dom * Qa;

apoly = /* polynomial defining 'a'. */

Qa = calib_make_Qa_dom (apoly);
...
calib_free_Qa_dom (Qa);
```

Let d be the degree of the polynomial defining the algebraic number a. The "values" of this Qa domain are polynomials in $Q[a]$ having degree at most $d-1$ stored in the `struct calib_Za_obj` object.

The `struct calib_Qa_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Qa::init():

```
void (*init) (const struct calib_Qa_dom * f,
 struct calib_Qa_obj * x);
```

Initialize Qa value object `x` to be a member of the Qa domain `f`, where:

`f` is the Qa domain performing this operation; and
`x` is the Qa value object to be initialized.

Qa::clear():

```
void (*clear) (struct calib_Qa_obj * x);
```

Clear out the given Qa value object `x` (freeing all memory it might hold), where:

`x` is the Qa value object to be cleared.

Qa::set():

```
void (*set) (struct calib_Qa_obj * rop,
const struct calib_Qa_obj * op);
```

Set `rop` to `op` in $Q(a)$, where:

`rop` is the Qa value object receiving the result; and
`src` is the source Qa value object.

Qa::set_si():

```
void (*set_si) (struct calib_Qa_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in $Q(a)$, where:

`rop` is the Qa value object receiving the result; and
`op` is the signed integer to convert into $Q(a)$ form.

Qa::set_z():

```
void (*set_z) (struct calib_Qa_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in $Q(a)$, where:

`rop` is the Qa value object receiving the result; and
`op` is the GMP integer to convert into $Q(a)$ form.

Qa::set_q():

```
void (*set_q) (struct calib_Qa_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in $Q(a)$, where:

`rop` is the Qa value object receiving the result; and
`op` is the GMP rational to convert into $Q(a)$ form.

Qa::set_var_power():

```
void (*set_var_power)
 (struct calib_Qa_obj * rop,
  int power);
```

Set `rop` to `a**power` in $Q(a)$ (a is the $Q(a)$ polynomial variable), where:

`rop` is the Qa value object receiving the result; and
`power` is the power to set (may be positive, zero or negative).

Qa::set_Za_q():

```
void (*set_Za_q)
 (struct calib_Qa_obj * rop,
  const struct calib_Za_obj * op1,
  mpq_srcptr op2);
```

Set `rop` to `op1 * op2` (`op1` is a Za value and `op2` is a GMP rational), where:

`rop` is the Qa value object receiving the result;
`op1` is the Za value to convert into Qa form; and
`op2` is a GMP rational multiplier.

The generator polynomials for `rop` and `op1` must have the same degree (so that this conversion can be done one coefficient at a time).

Qa::set_Zx():

```
void (*set_Zx)
 (struct calib_Qa_obj * rop,
  const struct calib_Zx_obj * op);
```

Set `rop` to `op` (`op` is a Zx polynomial whose variable becomes `a`), where:

`rop` is the Qa value object receiving the result;
`op` is the Zx polynomial to evaluate at `a`.

Polynomial `op` must have degree strictly less than the generator polynomial of the Qa field.

Qa::add():

```
void (*add) (struct calib_Qa_obj * rop,
const struct calib_Qa_obj * op1,
const struct calib_Qa_obj * op2);
```

Set `rop` to `op1 + op2` in $Q(a)$, where:

`rop` is the Qa value object receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Qa::sub():

```
void (*sub) (struct calib_Qa_obj * rop,
const struct calib_Qa_obj * op1,
const struct calib_Qa_obj * op2);
```

Set `rop` to `op1 - op2` in $Q(a)$, where:

`rop` is the Qa value object receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Qa::neg():

```
void (*neg) (const struct calib_Qa_dom * f,
mpq_ptr result,
mpq_srcptr op);
```

Set `rop` to `- op` in `Q(a)`, where:

`rop`	is the Qa value object receiving the result; and
`op`	is the operand being negated.

Qa::mul():

```
void (*mul) (struct calib_Qa_obj * rop,
const struct calib_Qa_obj * op1,
const struct calib_Qa_obj * op2);
```

Set `rop` to `op1 * op2` in $Q(a)$, where:

`rop`	is the Qa value object receiving the result;
`op1`	is the first operand; and
`op2`	is the second operand.

Qa::mul_si():

```
void (*mul_si) (struct calib_Qa_obj * rop,
   const struct calib_Qa_obj * op1,
   calib_si_t op2);
```

Set `rop` to `op1 * op2` in $Q(a)$, where:

`rop`	is the Qa value object receiving the result;
`op1`	is the first operand; and
`op2`	is the second operand.

Qa::mul_z():

```
void (*mul_z) (struct calib_Qa_obj * rop,
  const struct calib_Qa_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in $Q(a)$, where:

`rop`	is the Qa value object receiving the result;
`op1`	is the first operand; and
`op2`	is the second operand.

Qa::mul_q():

```
void (*mul_q) (struct calib_Qa_obj * rop,
  const struct calib_Qa_obj * op1,
  mpq_srcptr op2);
```

Set `rop` to `op1 * op2` in $Q(a)$, where:

`rop`	is the Qa value object receiving the result;
`op1`	is the first operand; and

`op2` is the second operand.

Qa::ipow():

```
void (*ipow) (struct calib_Qa_obj * rop,
 const struct calib_Qa_obj * op,
 int power);
```

Set `rop` to `op**power` in $Q(a)$, where:

`rop` is the Qa value object receiving the result;
`op` is the operand to exponentiate; and
`power` is the power to take (can be positive, negative or zero).

Qa::inv():

```
void (*inv) (struct calib_Qa_obj * rop,
const struct calib_Qa_obj * op);
```

Set `rop` to the multiplicative inverse of `op` in $Q(a)$, where:

`rop` is the Qa value object receiving the result; and
`op` is the operand to invert.

Qa::is_algint():

```
calib_bool
(*is_algint) (const struct calib_Qa_dom * dom);
```

Return TRUE if-and-only-if the given $Q(a)$ domain is an algebraic integer, where:

`dom` is the Qa domain to test.

Qa::algint_dom():

```
const struct calib_Qa_dom *
(*algint_dom) (const struct calib_Qa_dom * dom);
```

Return the Qa domain representing the algebraic integer corresponding to given domain `dom`, where:

`dom` is the Qa domain for which to get the corresponding algebraic integer domain.

Returns `dom` when `dom` is already an algebraic integer. Note: Do *NOT* free this domain, because it is owned by the given $Q(a)$ domain `dom`!

Qa::to_algint():

```
void (*to_algint) (struct calib_Qa_obj * rop,
      const struct calib_Qa_obj * op);
```

Set `rop` to be the same $Q(a)$ value as `op`, but represented in terms of the algebraic integer corresponding to `op`'s domain, where:

`rop` is the Qa value object receiving the result; and
`op` is the source Qa value.

Let D be the Qa domain of `op`. The domain of `rop` must either be identically D, or it must be the algebraic integer domain corresponding to D (e.g., as returned by `Qa::algint_dom()`).

Qa::from_algint():

```
void (*from_algint) (struct calib_Qa_obj * rop,
const struct calib_Qa_obj * op);
```

Set `rop` to be the same $Q(a)$ value as `op`, converting from the algebraic integer domain back to the original (possibly algebraic non-integer) domain, where:

`rop` is the Qa value object receiving the result; and
`op` is the source Qa value.

Let D be the Qa domain of `rop`. The domain of `op` must either be identically D, or it must be the algebraic integer domain corresponding to D (e.g., as returned by `Qa::algint_dom()`).

Qa::degree():

```
int (*degree) (const struct calib_Qa_obj * op);
```

Return the degree (in variable a of the $Q(a)$ generator polyomial) of the given element `op` of $Q(a)$, where:

`op` is the operand whose degree is to be returned.

Qa::zerop():

```
calib_bool
(*zerop) (const struct calib_Qa_obj * op);
```

Return 1 if-and-only-if `op` is identically zero and 0 otherwise, where:

`op` is the Qa value object to test for zero.

Qa::onep():

```
calib_bool
(*onep) (const struct calib_Qa_obj * op);
```

Return 1 if-and-only-if `op` is identically 1 and 0 otherwise, where:

`op` is the Qa value object to test for one.

Qa::set_genrep():

```
void (*set_genrep) (struct calib_Qa_obj * rop,
        const struct calib_genrep * op,
        const char * var);
```

Given a genrep `op` and the name `var` of the $Q(a)$ generator polynomial variable (as it appears in `op`), convert this genrep into a value in this $Q(a)$ domain, storing the result in `rop`, where:

`rop` is the Qa value object receiving the $Q(a)$ value;
`op` is the genrep being converted; and

`var` is the name of the algebraic number `a` appearing within genrep `op`.

Qa::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Qa_obj * op,
      const char * var);
```

Return a dynamically-allocated genrep representing the given Qa value `op`, using the given variable name `var` to represent the algebraic number a of $Q(a)$, where:

`op` is value from $Q(a)$ being converted to genrep form; and
`var` is the name of the algebraic number `a` as it should appear within the genrep returned.

Qa::bit_size():

```
size_t (*bit_size)
(const struct calib_Qa_obj * op);
```

Return the maximum "bit size" among all rational coefficients of the given Qa value `op` (the rational bit size is the number of significant bits in the product of the numerator and denominator), where:

`op` is the Qa value for which to return the bit size.

Qa::get_coeffs():

```
mpq_ptr (*get_coeffs)
(const struct calib_Qa_obj * op);
```

Return an array of GMP rationals containing the coefficients of the given Qa value `op`, where:

`op` is the Qa value for which to return the coefficients.

It is the caller's responsibility to free the returned array (of length `op->degree+1`).

Qa::set_coeffs():

```
void (*set_coeffs)
(struct calib_Qa_obj * rop,
 int degree,
 mpq_srcptr coeffs);
```

Set `rop` to be the $Q(a)$ value having the given degree `degree` and rational coefficients `coeffs`, where:

`rop` is the Qa value object receiving the result;
`degree` is the degree (in algebraic number `a`) of the value; and
`coeffs` is an array of `degree+1` rational coefficients.

It is permitted for `degree >= k`, where `k` is the degree of the generator polynomial `g(a)` defining this algebraic number domain (in which case the resulting value is reduced modulo `g(a)`).

Qa::get_Za_dom():

```
const struct calib_Za_dom *
```

```
(*get_Za_dom) (const struct calib_Qa_dom * f);
```

Return the $Z(a)$ version of this $Q(a)$ domain, where:

`f` is the Qa field/domain whose corresponding Za ring/domain is sought.

Note: Do *NOT* free this domain, because it is owned by the given $Q(a)$ domain `rp`!

16 Qax — The Polynomial Ring $Q(a)[x]$

CALIB provides the `Qax` domain, representing the ring $Q(a)[x]$, the univariate polynomials having coefficients that are an algebraic extension of the rationals. The “values” of this domain are represented by the following object:

```
/*
 * An instance of a polynomial in K[x], where K = Q(a).
 */

struct calib_Qax_obj {
int degree; /* Degree of polynomial */
int size; /* Size of coeff buffer */
const struct calib_Qax_dom *
dom; /* Qax domain of polymial */
struct calib_Qa_obj *
coeff; /* Coefficients of polynomial.  */
};
```

The CALIB `Qax` domain is constructed by specifying a `Qa` domain used to represent the coefficients of the corresponding Qax polynomials.

One may access CALIB’s `Qax` domain as follows:

```
#include "calib/Qax.h"
...
struct calib_Zx_obj * gpoly;
struct calib_Qa_dom * Qa;
        struct calib_Qax_dom * Qax;
struct calib_Qax_obj poly1, poly2;
...
gpoly = /* generator poly in Z[x] */
Qa = calib_make_Qa_dom (gpoly);
Qax = calib_make_Qax_dom (Qa);
...
Qax -> init (Qax, &poly1);
Qax -> init (Qax, &poly2);
...
Qax -> mul (Qax, &poly1, &poly1, &poly2);
...
Qax -> clear (&poly2);
Qax -> clear (&poly1);
calib_free_Qax_dom (Qax);
calib_free_Qa_dom (Qa);
```

The CALIB `Qax` domain supports the following settings:

```
/*
 * Which factorization algorithm to use (for square-free polynomials).
 */
```

```
enum calib_Qax_factor_method {
CALIB_QAX_FACTOR_METHOD_WEINBERGER_ROTHSCHILD,
CALIB_QAX_FACTOR_METHOD_NORM
};

/*
 * The "settings" object for the Qax domain.
 */

struct calib_Qax_settings {
/* Factorization method for square-free Qax polynomials. */
enum calib_Qax_factor_method factor_method;

/* Print initial factors during Weinberger-Rothschild? */
calib_bool WR_print_initial_factors;

/* Print lifted factors during Weinberger-Rothschild? */
calib_bool WR_print_lifted_factors;

/* Print trial factor combinations during Weinberger-Rothschild? */
calib_bool WR_print_trial_factor_combinations;
};

/*
 * Newly created Qax domains default to these settings.
 */

extern struct calib_Qax_settings calib_Qax_default_settings;
```

Each CALIB `Qax` domain has its own copy of these settings (consulted by the domain's operations):

```
struct calib_Qax_dom {
...
struct calib_Qax_settings settings;
...
};
```

These settings are initialized from `calib_Qax_default_settings` when the domain is constructed, but applications may alter these settings after construction, if desired.

The `struct calib_Qax_dom` object contains the following members (pointers to functions) that provide operations of the domain:

Qax::init():

```
void (*init) (const struct calib_Qax_dom * K_of_x,
 struct calib_Qax_obj * x);
```

Initialize the given Qax polynomial `x` to be the constant zero polynomial of the given Qax domain `K_of_x`, where:

`K_of_x` is the Qax ring/domain performing this operation; and
`x` is the polynomial to initialize.

Qax::init_degree():

```
void (*init_degree)
(const calib_Qax_dom * K_of_x,
 struct calib_Qax_obj * x,
 int degree);
```

Initialize the given Qax polynomial `x` to be the constant zero polynomial (while assuring that internal buffers are sufficiently large to hold a polynomial of up to the given `degree` without further allocation), where:

`K_of_x` is the Qax ring/domain performing this operation;
`x` is the polynomial to initialize; and
`degree` is the guaranteed minimimum degree polynomial that `dst` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Qax::alloc():

```
void (*alloc) (struct calib_Qax_obj * rop,
  int degree);
```

Force the given (already initialized) polynomial `rop` to have buffer space sufficient to hold a polynomial of at least the given `degree`, where:

`rop` is the polynomial whose allocation is to be adjusted; and
`degree` is the guaranteed minimum degree polynomial that `dst` will be able to hold (without further buffer allocation) upon successful completion of this operation.

Qax::clear():

```
void (*clear) (struct calib_Qax_obj * x);
```

Clear out the given polynomial `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x` is the polynomial to be cleared.

Qax::set():

```
void (*set) (struct calib_Qax_obj * rop,
const struct calib_Qax_obj * op);
```

Set `rop` to `op` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to copy.

Qax::set_si():

```
void (*set_si) (struct calib_Qax_obj * rop,
   calib_si_t op);
```

Set `rop` to `op` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op` is the signed integer value to set.

Qax::set_z():

```
void (*set_z) (struct calib_Qax_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op` is the GMP integer value to set.

Qax::set_q():

```
void (*set_q) (struct calib_Qax_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op` is the GMP rational value to set.

Qax::set_var_power():

```
void (*set_var_power)
 (struct calib_Qax_obj * rop,
  int power);
```

Set the polynomial `rop` to be `x**power` (`x` is the Qax polynomial variable), where:

`rop` is the polynomial receiving the result;
`power` is the power to set (must be non-negative).

Qax::set_Zx():

```
void (*set_Zx) (struct calib_Qax_obj * rop,
   const struct calib_Zx_obj * op);
```

Set `rop` to `op`, converting from Zx to Qax form, where:

`rop` the destination Qax polynomial / domain; and
`op` the source Zx polynomial.

Qax::add():

```
void (*add) (struct calib_Qax_obj * rop,
const struct calib_Qax_obj * op1,
const struct calib_Qax_obj * op2);
```

Set `rop` to `op1 + op2` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Qax::sub():

```
void (*sub) (struct calib_Qax_obj * rop,
const struct calib_Qax_obj * op1,
const struct calib_Qax_obj * op2);
```

Set `rop` to `op1 - op2` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Qax::neg():

```
void (*neg) (struct calib_Qax_obj * rop,
const struct calib_Qax_obj * op);
```

Set `rop` to `- op` in $Q(a)[x]$, where:

`rop` is the Qax polynomial receiving the result; and
`op` is the Qax polynomial being negated.

Qax::mul():

```
void (*mul) (struct calib_Qax_obj * rop,
const struct calib_Qax_obj * op1,
const struct calib_Qax_obj * op2);
```

Set `rop` to `op1 * op2` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

Qax::ipow():

```
void (*ipow) (struct calib_Qax_obj * rop,
 const struct calib_Qax_obj * op,
 int power);
```

Set `rop` to `op ** power` in $Q(a)[x]$, where:

`rop` is the polynomial receiving the result;
`op` is the polynomial to exponentiate; and
`power` is the power to take (must be >= 0).

Qax::dup():

```
struct calib_Qax_obj *
(*dup) (const struct calib_Qax_obj * op);
```

Return a dynamically-allocated Qax polynomial that is a copy of `op`, where:

`op` is the polynomial to be duplicated.

Qax::free():

```
void (*free) (struct calib_Qax_obj * poly);
```

Free the given dynamically-allocated polynomial `poly`, where:

`poly` is the polynomial to be freed.

This is equivalent to performing `Qax -> clear (poly);`, followed by `free (poly);`.

Qax::eval():

```
void (*eval) (struct calib_Qa_obj * rop,
 const struct calib_Qax_obj * poly,
 const struct calib_Qa_obj * value);
```

Evaluate polynomial `poly` at the given `value`, storing the result in `rop`, where:

`rop` is the Qa value object receiving the result;
`poly` is the polynomial to be evaluated; and
`value` is the Qa value at which to evaluate the polynomial.

Qax::eval_poly():

```
void (*eval_poly)
(struct calib_Qax_obj * rop,
 const struct calib_Qax_obj * poly,
 const struct calib_Qax_obj * value);
```

Evaluate polynomial `poly` at the given `value` (which is itself a polynomial in `K_of_X`), storing the result in `rop`, where:

`rop` receives the Qax polynomial result;
`poly` is the polynomial to be evaluated; and
`value` is the Qax polynomial at which to evaluate the polynomial.

Qax::div():

```
void (*div) (struct calib_Qax_obj * quotient,
struct calib_Qax_obj * remainder,
const struct calib_Qax_obj * a,
const struct calib_Qax_obj * b);
```

Polynomial division in $Q(a)[x]$, where:

`quotient` receives the quotient polynomial (may be NULL);
`remainder` receives the remainder polynomial (may be NULL);
`a` is the dividend polynomial; and
`b` is the divisor polynomial (may not be zero).

Division in $Q(a)[x]$ has the following properties:

- $a = quotient * b + remainder$
- $degree(remainder) < degree(b)$

Qax::gcd():

```
void (*gcd) (struct calib_Qax_obj * gcd,
const struct calib_Qax_obj * a,
const struct calib_Qax_obj * b);
```

Compute the greatest common divisor (GCD) of `a` and `b`, storing the result in `gcd`, where:

`gcd` receives the resulting GCD polynomial (always monic);
`a` is the first operand polynomial; and
`b` is the second operand polynomial.

The GCD is always monic unless a = b = 0.

Qax::extgcd():

```
void (*extgcd) (struct calib_Qax_obj * gcd,
   struct calib_Qax_obj * xa,
   struct calib_Qax_obj * xb,
   const struct calib_Qax_obj * a,
   const struct calib_Qax_obj * b);
```

The extended Euclidean algorithm. Compute polynomials `gcd`, `xa` and `xb` such that $gcd = a * xa + b * xb$, where:

`gcd` receives the resulting GCD polynomial (always monic);
`xa` receives the multiplier polynomial for `a`;
`xb` receives the multiplier polynomial for `b`;
`a` is the first operand polynomial; and
`b` is the second operand polynomial.

The result satisfies the following properties:

1. $degree(xa) < degree(b)$
2. $degree(xb) < degree(a)$

Qax::factor():

```
struct calib_Qax_factor *
(*factor) (const struct calib_Qax_obj * poly);
```

Factor the given polynomial `poly` into its irreducible factors over $Q(a)[x]$, returning a linked list of these factors, where:

`poly` is the Qax polynomial to be factored.

Except for an optional leading constant factor, all other factors are monic and irreducible.

Qax::factor_square_free():

```
struct calib_Qax_factor *
(*factor_square_free)
(const struct calib_Qax_obj * sqfpoly);
```

Given `sqfpoly` that is monic, square-free and of degree at least 1, factor it into irreducible factors, where

`sqfpoly` is the Qax polynomial to be factored.

This is a subroutine used by Qax::factor(). Two different methods are supported:

1. Weinberger, Rothschild, Factoring Polynomials Over Algebraic Number Fields, ACM Trans. Math. Softw., 1976, Vol 2, 335–350, `https://api.semanticscholar.org/CorpusID:6704708`.
2. The "norm" method (the default).

Qax::finish_sqf():

```
struct calib_Qax_factor *
(*finish_sqf) (
const struct calib_Qax_factor * sqfactors);
```

Given a list of monic, square-free polynomial `sqfactors`, "finish" the factorization by factoring each such factor into irreducible polynomials, returning a linked-list of these factors, where:

`sqfactors` is a linked-list of primitive, square-free Qax polynomial factors for which factorization into irreducibles is desired.

Qax::free_factors():

```
void (*free_factors)
(struct calib_Qax_factor * factors);
```

Free up the given list of Qax factors, where:

`factors` is a linked-list factors to be freed.

Qax::algint_dom():

```
const struct calib_Qax_dom *
(*algint_dom) (const struct calib_Qax_dom * dom);
```

Return the Qax domain representing the algebraic integer corresponding to given domain `dom`, where:

`dom` is the Qax domain for which to get the corresponding algebraic integer domain.

Returns `dom` when the coefficient field of `dom` is already an algebraic integer. Note: Do *NOT* free this domain, because it is owned by the given $Q(a)[x]$ domain `dom`!

Qax::to_algint():

```
void (*to_algint) (struct calib_Qax_obj * rop,
      const struct calib_Qax_obj * op);
```

Set `rop` to be the same $Q(a)[x]$ value as `op`, but coefficients represented in terms of the algebraic integer corresponding to `op`'s coefficient domain, where:

`rop` is the Qax polynomial receiving the result; and
`op` is the source Qax polynomial.

Let D be the Qax domain of `op`. The domain of `rop` must either be identically D, or it must be the algebraic integer domain corresponding to D (e.g., as returned by `Qax::algint_dom()`).

Qax::from_algint():

```
void (*from_algint) (struct calib_Qax_obj * rop,
const struct calib_Qax_obj * op);
```

Set `rop` to be the same $Q(a)[x]$ polynomial as `op`, converting the coefficients from the algebraic integer domain back to the original (possibly algebraic non-integer) domain, where:

`rop` is the Qax polynomial receiving the result; and
`op` is the source Qax polynomial.

Let D be the Qax domain of `rop`. The domain of `op` must either be identically D, or it must be the algebraic integer domain corresponding to D (e.g., as returned by `Qax::algint_dom()`).

Qax::zerop():

```
calib_bool
(*zerop) (const struct calib_Qax_obj * op);
```

Return 1 if-and-only-if the given Qax polynomial is identically zero and 0 otherwise, where:

`op` is the Qax polynomial to test for zero.

Qax::onep():

```
calib_bool
(*onep) (const struct calib_Qax_obj * op);
```

Return 1 if-and-only-if the given Qax polynomial is identically one and 0 otherwise, where:

`op` is the Qax polynomial to test for one.

Qax::set_genrep():

```
void (*set_genrep) (struct calib_Qax_obj * rop,
        const struct calib_genrep * op,
        const char * xvar,
        const char * avar);
```

Compute a Qax polynomial obtained from the given genrep `op`, interpreting `xvar` to be the name of the variable used by the Qax polynomial and `avar` to be the name of the variable used by the Qa coefficients, storing the result in `rop`, where:

`rop` is the Qax polynomial receiving the result;
`op` is the genrep to convert into Qax polynomial form;
`xvar` is the variable name (appearing within genrep `op`) that is to be interpreted as the polynomial variable in Qax; and
`avar` is the variable name (appearing within genrep `op`) that is to be interpreted as the variable used by the Qa coefficients.

Qax::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_Qax_obj * op,
```

```
          const char * xvar,
          const char * avar);
```

Return a dynamically-allocated genrep corresponding to the given Qax polynomial `op`, using `xvar` as the name of the Qax polynomial variable and `avar` as the name of the Qa coefficient variable within the returned genrep, where:

`op` is the Qax polynomial to convert into genrep form;

`xvar` is the variable name to use in the genrep for the polynomial variable of Qax; and

`avar` is the variable name to use in the genrep for the Qa coefficients.

Qax::factors_to_genrep():

```
struct calib_genrep *
(*factors_to_genrep)
(const struct calib_Qax_factor * factors,
 const char * xvar,
 const char * avar);
```

Return a dynamically-allocated genrep corresponding to the given list of Qax `factors`, using `xvar` as the name of the Qax polynomial variable and `avar` as the name of the Qa coefficient variable within the returned genrep, where:

`factors` is the list of Qax polynomial factors to convert into genrep form; and

`xvar` is the variable name to use in the genrep for the polynomial variable of Qax;

`avar` is the variable name to use in the genrep for the Qa coefficients.

Qax::bit_size():

```
size_t (*bit_size)
(const struct calib_Qax_obj * op);
```

Return the maximum "bit size" among all rational coefficients of the given Qax polynomial `op` (the rational bit size is the number of significant bits in the product of the numerator and denominator), where:

`op` is the Qax polynomial for which to return the bit size.

Qax::factors_bit_size():

```
size_t (*bit_size)
(const struct calib_Qax_factor * factors);
```

Return the maximum "bit size" among all rational coefficients of all the given list `factors` of Qax factors (the rational bit size is the number of significant bits in the product of the numerator and denominator), where:

`factors` is a linked-list of Qax polynomial factors for which to return the bit size.

Qax::get_Zax_dom():

```
const struct calib_Zax_dom *
(*get_Zax_dom) (const struct calib_Qax_dom * K_of_x);
```

Return the $Z(a)[x]$ version of the given $Q(a)[x]$ domain `K_of_x`, where:

`K_of_x` is the Qax polynomial domain whose corresponding Zax polynomial domain is sought.

Note: Do *NOT* free this domain, because it is owned by the given $Q(a)[x]$ domain `K_of_x`!

17 rat — The Rational Functions $Z[x,y,z]/Z[x,y,z]$

CALIB provides to `rat` domain, representing the quotient field R/R — the rational functions — where R is the ring of multi-variate polynomials with integer coefficients.

The "values" of this domain are represented by the following object:

```
struct calib_rat_obj {
const struct calib_rat_dom *
dom; /* Rational function domain */
struct calib_Zxyz_obj numer; /* Numerator polynomial */
struct calib_Zxyz_obj denom; /* Denominator polynomial */
};
```

The CALIB `rat` domain is constructed by specifying a `Zxyz` domain to represent the numerator and denominator of the corresponding rational functions.

The `calib/rat.h` header defines the following additional objects:

```
/*
 * An object to represent a set of variable substitutions,
 * each variable being replaced with a given rational expression.
 */

struct calib_subst_list {
int num_subst; /* Number of substitutions */
struct calib_rsubst * subst; /* Array of substitutions */
};

/*
 * An object to represent a rational expression to substitute
 * for a given variable.
 * In order to accomodate variables that are not part of the polynomial
 * domain of `val', the `varname' field can be the name of an arbitrary
 * variable.  Convention:
 *
 * ((var >=  0) AND (varname EQ NULL)) OR
 * ((var EQ -1) AND (varname NE NULL))
 */

struct calib_rsubst {
int var; /* Variable number to be substituted */
/* (or -1) */
const char * varname;
/* Variable name (or NULL) */
struct calib_rat_obj val; /* Rational value to replace */
/* var with */
};
```

One may access CALIB's `rat` domain as follows:

```
#include "calib/rat.h"
```

```
#include "calib/Zxyz.h"
...
        const struct calib_Zxyz_dom * Zxyz;
const struct calib_rat_dom * rat;
struct calib_rat_obj ratvar;
const char * varnames [3] = {"x", "y", "z"};
...
Zxyz = calib_make_Zxyz_dom (3, varnames);
rat = calib_make_rat_dom (Zxyz);
...
rat -> init (rat, &ratvar);
...
rat -> canonicalize (&ratvar);
...
rat -> print (&ratvar);
...
rat -> clear (&ratvar);
...
calib_free_rat_dom (rat);
calib_free_Zxyz_dom (Zxyz);
```

The `struct calib_rat_dom` object contains the following members (pointers to functions) that provide operations of the domain:

rat::init():

```
void (*init) (const struct calib_rat_dom * dom,
 struct calib_rat_obj * x);
```

Initialize the given rat object `x`, where:

`R_of_x`	is the rat field/domain performing this operation;
`x`	is the rational object to initialize.

rat::clear():

```
void (*clear) (struct calib_rat_obj * x);
```

Clear out the given rational object `x` (freeing all memory it might hold and returning it to the constant value of zero), where:

`x`	is the rational object to be cleared.

rat::set():

```
void (*set) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op);
```

Set `rop` to `op` in the “rat” domain, where:

`rop`	is the rational function receiving the result; and
`op`	is the rational function to copy.

rat::set_si():

```
void (*set_si) (struct calib_rat_obj * rop,
```

```
        calib_si_t op);
```

Set `rop` to `op` in the “rat” domain, where:

`rop` is the rational function receiving the result; and
`op` is the integer value to set.

rat::set_z():

```
void (*set_z) (struct calib_rat_obj * rop,
  mpz_srcptr op);
```

Set `rop` to `op` in the “rat” domain, where:

`rop` is the rational function receiving the result; and
`op` is the GMP integer value to set.

rat::set_q():

```
void (*set_q) (struct calib_rat_obj * rop,
  mpq_srcptr op);
```

Set `rop` to `op` in the “rat” domain, where:

`rop` is the rational function receiving the result; and
`op` is the GMP rational value to set.

rat::set_var_power():

```
void (*set_var_power)
 (struct calib_rat_obj * rop,
  int var,
  int power);
```

Set `rop` to `var ** power` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`var` is the index of the variable; and
`power` is the power to set (may be positive, zero or negative).

rat::add():

```
void (*add) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op1,
const struct calib_rat_obj * op2);
```

Set `rop` to `op1 + op2` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

rat::sub():

```
void (*sub) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op1,
const struct calib_rat_obj * op2);
```

Set `rop` to `op1 - op2` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

rat::neg():

```
void (*neg) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op);
```

Set `rop` to `- op` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`op` is the operand to negate.

rat::mul():

```
void (*mul) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op1,
const struct calib_rat_obj * op2);
```

Set `rop` to `op1 * op2` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

rat::mul_z():

```
void (*mul_z) (struct calib_rat_obj * rop,
  const struct calib_rat_obj * op1,
  mpz_srcptr op2);
```

Set `rop` to `op1 * op2` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`op1` is the first (rational function) operand; and
`op2` is the second (GMP integer) operand.

rat::mul_q():

```
void (*mul_q) (struct calib_rat_obj * rop,
  const struct calib_rat_obj * op1,
  mpq_srcptr op2);
```

Set `rop` to `op1 * op2` in the “rat” domain, where:

`rop` is the rational function receiving the result;
`op1` is the first (rational function) operand; and
`op2` is the second (GMP rational) operand.

rat::ipow():

```
void (*ipow) (struct calib_rat_obj * rop,
 const struct calib_rat_obj * op,
 int power);
```

Set `rop` to `op ** power` in the "rat" domain, where:

`rop` is the rational function receiving the result;
`op` is the rational function to exponentiate; and
`power` is the power to take (may be positive, zero or negative).

rat::inv():

```
void (*inv) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op);
```

Set `rop` to `1 / op` in the "rat" domain, where:

`rop` is the rational function receiving the result;
`op` is the operand to invert.

rat::div():

```
void (*div) (struct calib_rat_obj * rop,
const struct calib_rat_obj * op1,
const struct calib_rat_obj * op2);
```

Set `rop` to `op1 / op2` in the "rat" domain, where:

`rop` is the rational function receiving the result;
`op1` is the first operand; and
`op2` is the second operand.

rat::canonicalize():

```
void (*canonicalize (struct calib_rat_obj * op);
```

Modify (if necessary) the given rational object to be in "canonical" form:

- All common factors between the numerator and denominator are removed,
- The leading coefficient of the denominator is positive,

where:

`op` is the rational object to be canonicalized.

rat::factor():

```
struct calib_Zxyz_factor *
(*factor (const struct calib_rat_obj * op);
```

Factor `op`, returning a list of factors, where:

`op` is the rational object to be factored.

Note: This returns a list of `struct calib_Zxyz_factor` objects for which the denominator's factors have *negative* multiplicies. The `Zxyz::factors_to_genrep` routine properly handles such negative multiplicities.

rat::substitute_with_varmap():

```
void (*substitute_with_varmap)
(struct calib_rat_obj * rop,
```

```
const struct calib_rat_obj * op,
const int * varmap,
struct calib_subst_list * slist);
```

Efficiently perform a set `slist` of variable substitutions (`struct calib_rsubst` objects) each having the form:

```
var = subst_rat
```

to the given rational expression `op`, storing the result into `rop`. The subst_rat values are first transformed from one `Zxyz` ring to another according to the given `varmap` — an array indexed by variables appearing in the subst_rat values being substituted. Let $j = varmap[i]$. There are two cases:

- $j < 0$: variable i must not appear in any of the subst_rat values being substituted;
- $j >= 0$: variable i in the subst_rat values maps to variable j of `op` and `rop`.

The parameters are as follows:

`rop` receives the substitution result;
`op` is the rational object to which the substitutions are to be applied;
`varmap` defines variable transformations between the subst_rat expressions and the `op` and `rop` rings as described above; and
`slist` is the list of substitutions to perform.

rat::free_slist():

```
void (*free_slist) (struct calib_subst_list * slist);
```

Free the given list of rational substitutions, where:

`slist` defines an array of rational substitutions to free.

rat::zerop():

```
calib_bool
(*zerop) (const struct calib_rat_obj * op);
```

Return 1 if-and-only-if the given rational function is identically zero and 0 otherwise, where:

`op` is the rational function to test for zero.

rat::onep():

```
calib_bool
(*onep) (const struct calib_rat_obj * op);
```

Return 1 if-and-only-if the given rational function is identically one and 0 otherwise, where:

`op` is the rational function to test for one.

rat::set_genrep():

```
void (*set_genrep) (struct calib_rat_obj * rop,
        const struct calib_genrep * op);
```

Compute a rational function obtained from the given genrep `op`, storing the result in `rop`. Use the domain of `rop` to map variable names in `op` to variable numbers in the resulting rational function, where:

`rop` receives the resulting rational function; and
`op` is the genrep to convert into rational function form.

rat::to_genrep():

```
struct calib_genrep *
(*to_genrep) (const struct calib_rat_obj * op);
```

Return a dynamically-allocated genrep corresponding to the given rational function `op` (whose rat domain provides variable names to use for each variable number), where:

`op` is the rational function to convert into genrep form.

rat::print():

```
void (*print) (const struct calib_rat_obj * op);
```

Print the given rational expression (in Maxima input syntax) terminated with a newline, where:

`opt` is the rational expression to be printed.

rat::print_nnl():

```
void (*print_nnl) (const struct calib_rat_obj * op);
```

Print the given rational expression (in Maxima input syntax) with no terminating newline, where:

`opt` is the rational expression to be printed.

18 The "calib/calib.h" Header.

The "calib/calib.h" header is a convenient way to include the bulk of the CALIB library.
Synopsis of "calib/calib.h":

```
#include "calib/genrep.h"
#include "calib/GFpk.h"
#include "calib/GFpkx.h"
#include "calib/prompt.h"
#include "calib/Qa.h"
#include "calib/Qax.h"
#include "calib/Qx.h"
#include "calib/rat.h"
#include "calib/shutdown.h"
#include "calib/types.h"
#include "calib/Za.h"
#include "calib/Zax.h"
#include "calib/Zp.h"
#include "calib/Zpx.h"
#include "calib/Zx.h"
#include "calib/Zxyz.h"
```

19 The "calib/cputime.h" Header.

The "calib/cputime.h" header provides facilities for measuring the CPU time usage of code regions.

```
typedef unsigned long calib_cpu_time_t;
```

The type used by CALIB to represent elapsed CPU time. This is in units of 1/100 of a second.

calib_get_cpu_time:

```
calib_cpu_time_t
calib_get_cpu_time (void);
```

Returns the total elapsed CPU time consumed by the calling process (and all of its waited-for children, both user and system time).

calib_get_delta_cpu_time:

```
calib_cpu_time_t
calib_get_delta_cpu_time (calib_cpu_time_t * time_in_out);
```

Returns the delta CPU time since the previous measurement `*time_in_out`, while updating `*time_in_out` to be the current elapsed CPU time, where

`time_in_out` points to a `calib_cpu_time_t` containing the previous elapsed CPU time, and is updated to contain the current elapsed CPU time.

calib_convert_cpu_time:

```
void calib_convert_cpu_time (calib_cpu_time_t time,
char * buf);
```

Convert CPU time into printable text, where:

`time` is a CPU time (or delta CPU time); and

`buf` is a character buffer to receive the textual time in CPU seconds, to a resolution of 2 decimal places.

calib_convert_delta_cpu_time:

```
void calib_convert_delta_cpu_time (
char * buf,
calib_cpu_time_t * time_in_out);
```

Combines the functionality of `calib_get_delta_cpu_time` and `calib_convert_cpm_time`, where:

`buf` is a character buffer to receive the textual delta CPU time (in seconds, to 2 decimal places); and

`time_in_out` points to a `calib_cpu_time_t` containing the previous elapsed CPU time, and is updated to contain the current elapsed CPU time.

20 The "calib/fatal.h" Header.

The `"calib/fatal.h"` header contains facilities to detect and report *fatal* software errors. Fatal software errors report the source file and line number to stderr and then call `abort()` to produce a core file.

The following macros are provided:

```
CALIB_FATAL_ERROR;
```

Produces an immediate fatal error at this source file and line number.

```
CALIB_FATAL_ERROR_IF (condition);
```

Produces an immediate fatal error at this source file and line number if the given condition is true.

21 The "calib/gmpmisc.h" Header.

The "calib/gmpmisc.h" header contains useful GMP extensions that are unavailable in older versions of GMP. To prevent name clashes with newer versions of GMP that provide similar functions, we adopt CALIB-style names for these functions.

There are three broad classes of GMP functions provided by CALIB:

- Functions we wish that GMP provided;
- Allocation, initialization, clearing, freeing, copying and printing of *vectors* of GMP integers and rationals; and
- Functions to compute the "bit size" of GMP integers, rationals and vectors thereof.

calib_mpq_set_z_z:

```
void calib_mpq_set_z_z (mpq_ptr rop,
   mpz_srcptr op1,
   mpz_srcptr op2);
```

Set rop to op1 / op2.

calib_mpq_mul_si:

```
void calib_mpq_mul_si (mpq_ptr rop,
  mpq_srcptr op1,
  calib_si_t op2);
```

Set rop to op1 * op2.

calib_mpq_mul_z:

```
void calib_mpq_mul_z (mpq_ptr rop,
 mpq_srcptr op1,
 mpz_srcptr op2);
```

Set rop to op1 * op2.

calib_mpq_div_si:

```
void calib_mpq_div_si (mpq_ptr rop,
  mpq_srcptr op1,
  calib_si_t op2);
```

Set rop to op1 / op2.

calib_mpq_div_z:

```
void calib_mpq_div_z (mpq_ptr rop,
 mpq_srcptr op1,
 mpz_srcptr op2);
```

Set rop to op1 / op2.

calib_new_Z_vector:

```
mpz_ptr calib_new_Z_vector (size_t n);
```

Returns a dynamically-allocated and initialized (to zero) vector of n GMP integers.

calib_free_Z_vector:

```
void calib_free_Z_vector (mpz_ptr p, size_t n);
```

Clears and frees dynamically-allocated vector p of n GMP integers.

calib_init_Z_vector:

```
void calib_init_Z_vector (mpz_ptr p, size_t n);
```

Initialize the given vector `p` of `n` GMP integers.

calib_clear_Z_vector:

```
void calib_clear_Z_vector (mpz_ptr p, size_t n);
```

Clear the given vector `p` of `n` GMP integers.

calib_copy_Z_vector:

```
void calib_copy_Z_vector (mpz_ptr dst, mpz_srcptr src, size_t n);
```

Copy vector `src` of `n` GMP integers into vector `dst`. These vectors must not overlap.

calib_print_Z_vector:

```
void calib_print_Z_vector (mpz_srcptr p, size_t n);
```

Print vector `p` of `n` GMP integers, separated by spaces and terminated with a newline.

calib_new_Q_vector:

```
mpq_ptr calib_new_Q_vector (size_t n);
```

Returns a dynamically-allocated and initialized (to zero) vector of `n` GMP rationals.

calib_free_Q_vector:

```
void calib_free_Q_vector (mpq_ptr p, size_t n);
```

Clears and frees dynamically-allocated vector `p` of `n` GMP rationals.

calib_init_Q_vector:

```
void calib_init_Q_vector (mpq_ptr p, size_t n);
```

Initialize the given vector `p` of `n` GMP rationals.

calib_clear_Q_vector:

```
void calib_clear_Q_vector (mpp_ptr p, size_t n);
```

Clear the given vector `p` of `n` GMP rationals.

calib_copy_Q_vector:

```
void calib_copy_Q_vector (mpq_ptr dst, mpq_srcptr src, size_t n);
```

Copy vector `src` of `n` GMP rationals into vector `dst`. These vectors must not overlap.

calib_print_Q_vector:

```
void calib_print_Q_vector (mpq_srcptr p, size_t n);
```

Print vector `p` of `n` GMP rationals, separated by spaces and terminated with a newline.

calib_mpz_bit_size:

```
size_t calib_mpz_bit_size (mpz_srcptr z);
```

Return the bit size of GMP integer `z`. (The bit size is the number of significant bits of the absolute value.)

calib_mpz_bit_size_vector:

```
size_t calib_mpz_bit_size (mpz_srcptr op, size_t n);
```

Return the maximum bit size among all `n` elements of vector `op` of GMP integers.

calib_mpq_bit_size:

```
size_t calib_mpq_bit_size (mpq_srcptr q);
```

Return the bit size of GMP rational q. (The bit size is the number of significant bits in the absolute value of the product of the numerator and denominator.)

calib_mpq_bit_size_vector:

```
size_t calib_mpq_bit_size (mpq_srcptr op, size_t n);
```

Return the maximum bit size among all n elements of vector op of GMP rationals.

22 The "calib/lll.h" Header.

The "calib/lll.h" header currently defines a single function:

```
int _calib_LLL (mpz_ptr res,
        mpz_srcptr b,
        int nvec,
        int vsize);
```

This is a CALIB-provided implementation of the Lenstra, Lenstra, Lovász (LLL) lattice-basis reduction algorithm. This implementation is not yet ready for use. (The leading underscore will disappear when this changes. In the mean time, we encourage the use of FPLLL. CALIB's internal LLL algorithmitm will never beat the performance of FPLLL, which is very large and complex.)

The basis vector arguments to this routine consist of a sequence of `nvec` integer vectors, each having `vsize` elements. The `i`-th vector begins at element `i*vsize` of the array.

The parameters are as follows:

`res`	array that receives the reduced lattice-basis vectors;
`b`	array containing the lattic-basis vectors to reduce;
`nvec`	number of basis vectors; and
`vsize`	number of elements in each basis vector.

Returns zero upon success and a non-zero code upon failure.

23 The "calib/logic.h" Header.

The "calib/logic.h" header contains macros that smooth out some of the C programming language's sharp edges:

```
#define NOT !
#define AND &&
#define OR ||
#define EQ ==
#define NE !=

#define FALSE 0
#define TRUE 1

#ifndef NULL
 #define NULL 0
#endif
```

For example, it is a common mistake to type = where == was intended. Many hours were wasted trying to find a bug whereing |= was typed instead of !=. The CALIB source code always uses EQ and NE instead, and doing likewise in your own coding conventions can save heartache. Similarly, it is common to accidentally type & instead of &&, and | instead of ||. Learning to always use AND, OR and NOT can similarly avoid such problems.

24 The "calib/new.h" Header.

The "calib/new.h" header contains the following macros and functions that make allocating memory much friendly and more type-safe:

```
#define CALIB_NEW(T) ((T *) _calib_new (sizeof (T)))

#define CALIB_NEWA(N,T) ((T *) _calib_new ((N) * sizeof (T)))

#define CALIB_FREE(p) \
do { if ((p) NE NULL) { free ((void *) (p)); } } while (FALSE)

extern void * _calib_new (size_t nbytes);
```

CALIB_NEW(T) dynamically allocates a single (uninitialized) object of type T. CALIB_NEWA(N, T) dynamically allocates an array (uninitialized) of N objects of type T. CALIB_FREE(p) checks for NULL before calling free(). The _calib_new function catches the out-of-memory condition (by printing an error message and calling exit(1)).

25 The "calib/prompt.h" Header.

The "calib/prompt.h" header contains a single function:

```
void calib_prompt (const char * str);
```

It prints out the given string as a “prompt” — but only if stdin is a tty (i.e., we are receiving interactive input from a user, not input from a file). This function is used by various CALIB test programs, and could perhaps be useful in other contexts.

26 The "calib/random.h" Header.

The "calib/random.h" header contains various facilities for generating random numbers. The random numbers generated herein are far from being cryptographically sound — they are used only for various algebraic algorithms (i.e., distinct-degree factorization in Zp[x]) that do not require a high degree of randomness. Only one initial seed is supported, and no randomization by wall time or similar means is provided.

The following random state object is defined:

```
struct calib_Random {
calib_int32u lo;
calib_int32u hi;
double normal_val2; /* Cache 2nd normal deviate */
int normal_flag; /* val2 is valid iff flag is TRUE */
};
```

calib_random_init:

```
void calib_random_init (struct calib_Random * state);
```

Initialize the given random state object `state`.

calib_random:

```
double calib_random (struct calib_Random * state);
```

Return a double-precision value uniformly distributed in the half-open interval `[0.0, 1.0)`.

calib_random_normal:

```
double calib_random_normal (struct calib_Random * state);
```

Return a double-precision value normally distributed with mean `0.0` and variance `1.0`.

calib_random_u32:

```
calib_int32u calib_random_u32 (struct calib_Random * state);
```

Return a uniformly distributed unsigned 32-bit value.

calib_random_u64:

```
calib_int64u calib_random_u64 (struct calib_Random * state);
```

Return a uniformly distributed unsigned 64-bit value.

27 The "calib/shutdown.h" Header.

The `"calib/shutdown.h"` header provides a single function:

```
void calib_shutdown (void);
```

When an application is finished using the CALIB library, it can call this function to free up all memory that is statically held by the CALIB library.

Applications should be careful to *not* call this function while CALIB objects requested by the application still exist, as this function may cause these objects to become invalid and inconsistent.

If the application has freed all its CALIB objects and then called `calib_shutdown()`, CALIB is specifically designed so that the application can start using CALIB once again. This can serve to *release* all of CALIB's statically-held memory back to the memory heap. (Of course the run time memory heap implementation may or may not release this memory back to the operating system.)

The `calib_shutdown()` function is intended to assist in eliminating memory leaks from applications that use CALIB, and perhaps also in reducing memory usage caused by "intermediate expression swell" at points where CALIB is otherwise quiescent.

28 Sample Applications Using CALIB

CALIB provides several applications serving as examples of how to use CALIB to solve various symbolic computational problems.

28.1 combdist

The `combdist` application computes a closed-form formula giving the centroid distance (squared) for general 3-toothed comb facets of the Traveling Salesman Polytope $TSP(n)$. Let $G = (V, E)$ be the complete graph with $n = |V|$ vertices. Let $m = |E| = n(n-1)/2$. Let T be the set of all incidence vectors denoting subsets of E that form Hamiltonian cycles (tours) of G. Then polytope $TSP(n)$ is the convex hull of T.

The centroid C of $TSP(n)$ is defined to be the mean of all incidence vectors T.

Let A be the affine hull of $TSP(n)$. ($TSP(n)$ satisfies `n` equations requiring $degree(v) = 2$ for each vertex v.)

It is well-known that the *k-toothed comb inequalities* define facets of $TSP(n)$ for all odd $k >= 3$. A general 3-toothed comb inequality can be described by partitioning V into 8 mutually-disjoint sets: $B1, T1, B2, T2, B3, T3, H, O$, where Bi, Ti are the "base" and "tip" of tooth i; H are the handle vertices outside of any teeth; and O are the remaining vertices completely outside the comb. A valid comb inequality requires Bi and Ti to have at least one vertex each. H and O are permitted to be empty. We define the following parameters: $b1 = |B1|$, $t1 = |T1|$, $b2 = |B2|$, $t2 = |T2|$, $b3 = |B3|$, $t3 = |T3|$, $h = |H|$, and $o = |O|$.

Let $B1, T1, B2, T2, B3, T3, H, O$ be a partition of V, with $b1, t1, b2, t2, b3, t3, h, o$ defined correspondingly as above such that $b1, t1, b2, t2, b3, t3 >= 1$. This partition defines a unique 3-toothed comb inequality Q. Let F be hyperplane bounding Q. The centroid distance d is defined to be the shortest Euclidean distance between C and (F intersect A).

The centroid distance d is considered to be an "indicator" for the "strength" of the corresponding inequality Q. (Smaller centroid distance d indicate "stronger" inequalities: they cut more deeply into the polytope and exclude more volume from the feasible region than weaker inequalities having larger centroid distance.)

The `combdist` application computes a symbolic closed-form for the centroid distance (squared) as a function of $b1, t1, b2, t2, b3, t3, h, o$.

The computation is structured as follows. Let y be the point in (F intersect A) closest to C. The partition defines 8 classes of vertices, giving rise to 36 classes of edges. (For example, the class of edges having one vertex in $T3$ and the other vertex in H. The 36 edge classes form a parition of E.) Let J be one such edge class. By symmetry of $TSP(n)$ we have $y[e1] = y[e2]$ for all $e1, e2$ in J. It suffices, therefore to consider a point x in $R^3 6$ such that $y[e1] = y[e2] = x[J]$.

We start with a system of 9 equations over x: eight "degree 2" equations (one per vertex class), and the equation corresponding to the 3-toothed comb inequality. (The coefficients of these equations are polynomials in the 8 parameters $b1, t1, b2, t2, b3, t3, h, o$.) Use Gaussian elimination on this 9x36 system of equations to solve for 9 of the x variables in terms of the remaining 27. (The solution of this system represents those points x that reside in (F intersect A).) Now write down the formula for centroid distance squared (as a function of the 36 x variables and 8 parameters). Use the solution of the previous system of equations to eliminate 9 of the x variables. This squared distance is minimized when its partial derivative is zero with respect to each of the 27 remaining x variables. We therefore construct a 27x27 system of equations requiring each of these 27 partial derivatives be zero. (The coefficients of this system are rational functions of the 8 parameters. The coefficients become polynomials after clearing denominators within each row of the system.) Now use Gaussian elimination to solve this 27x27 system of equations. This gives a unique solution for the 27 remaining x variables. Substituting these into the first linear system gives unique values for the other 9 x variables. Substituting these into the symbolic "distance squared" formula yields the final closed-form we seek: the centroid distance (squared) as a function of the 8 parameters.

One must be careful when choosing pivots during Gaussian elimination for both of these linear systems. One should avoid any pivot for which feasible values of the 8 parameters yield a pivot value of zero. Given that these pivot elements are polynomials in the 8 parameters, this is equivalent to Hilbert's Tenth Problem (which was proven to be unsolvable by Matiyasevich). That this zero-pivot-recognition problem is unsolvable *in general* does not mean that algorithms cannot correctly solve *some* instances automatically. The `combdist` program contains an algorithm `hilbert_10_heuristic` that returns `TRUE` if-and-only-if no feasible set of parameter values can make the given pivot polynomial be zero. This heuristic makes execellent use of CALIB's facilities (especially factorization in `Zxyz`) to obtain these answers. There is just one pivot selection sub-problem for which this heuristic is unable to identify any of the pivot candidates as being "safe;" the code uses a manually-selected pivot in this case. (It is a large polynomial for which we have neither a proof of non-zeroness, nor a feasible set of parameter values that zero it.)

One must also beware of the special cases $h = 0$ and/or $o = 0$. The corresponding vertex sets are empty in these cases, and there are therefore no incident edges. This causes the corresponding degree-2 equation to effectively become $0 = 2$. One should therefore be reluctant to accept the general solution described above as being valid for these special cases. The `comdist` application therefore repeats the computation for each of the three special cases: $h = 0$ and $o > 0$; $h > 0$ and $o = 0$; and $h = 0$ and $o = 0$. The offending vertex/edge classes are omitted, as well as the offending degree-2 equation(s). In all three cases, the special solution is found to match that obtained by setting $h = 0$ and/or $o = 0$ in the general solution. The general solution is therefore truly general, and represents the correct centroid distance squared for every valid 3-toothed TSP comb inequality.

For $n <= 12$ it is practical to recursively enumerate the incidence vectors of all tours, allowing centroid distances to be computed "from first principles" via quadratic programming. For all 3-toothed combs tested, the closed-form produces centroid distances matching those obtained via quadratic programming.

28.2 ratint

The `ratint` application demonstrates CALIB being used to perform a very "traditional" computer algebra task — indefinite integration of rational functions of a single variable. (Integration of functions in $Z[x]/Z[x]$.) It reads a sequence of expressions from stdin that must have the following format:

```
integrate (expr, var);
```

The `expr` must be a rational function over the single variable `var`. It displays the original problem together with its solution. (It also checks the correctness of the solution by differentiating it and comparing it with the original integrand — printing a stern warning if they do not match.)

`ratint` accepts the following command line arguments:

`-l` Factor arguments of generated $log()$ terms.
`-p` Factor polynomial part of solution.
`-r` Factor generated rational function term.
`-t` Enable some tracing.

`ratint` solves this problem using the following classic computer algebra techniques:

1. Partial-fraction decomposition over the square-free factorization of the denominator.
2. Hermite reduction of integrands whose denominators are not square-free.
3. Trager's method for integrating R/S (R and S are primitive, square-free, relatively prime polynomials with $deg(R) < deg(S)$). Trager's method splits the factors of S in an "optimal" way that does not introduce any algebraic number constant multipliers on the generated $log()$ terms.
4. A traditional method for finding $atan()$ terms (possibly involving square-roots of integers).

It does *not* yet attempt to split factors Si of denominator S having the form $a * x^(2k) + b * x^k + c$ into separate partial-fractions, as these could contain square-roots of integers and be subject to additional algebraic decomposition which this simple machinery is not prepared to handle. (It does not even attempt this when $k = 1$, although this special case generates final $log()$ terms containing square-roots of integers having no further need of algebraic processing. Extending `ratint` to handle this $k = 1$ case is left as an interesting exercise for those wishing to experiment further with CALIB.)

Function Index

C

G

Q

R

Z

Index

C

F

G

R

S

Z