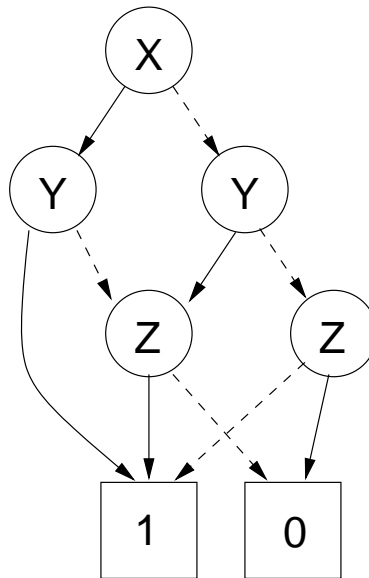


BuDDy: Binary Decision Diagram package  
Release 2.2

Jørn Lind-Nielsen  
IT-University of Copenhagen (ITU)  
e-mail: [buddy@itu.dk](mailto:buddy@itu.dk)  
November 9, 2002





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Acknowledgements . . . . .	1
<b>2</b>	<b>Users Guide</b>	<b>3</b>
2.1	Getting BuDDy . . . . .	3
2.2	Installing . . . . .	3
2.3	Compiling . . . . .	3
2.4	Programming with BuDDy . . . . .	3
2.4.1	More Examples . . . . .	4
2.5	Variable sets . . . . .	4
2.6	Dynamic Variable Reordering . . . . .	5
2.7	Error Handling . . . . .	6
2.8	The C++ interface . . . . .	6
2.9	Finite Domain Blocks . . . . .	7
2.10	Boolean Vectors . . . . .	7
2.10.1	C++ Interface . . . . .	9
<b>3</b>	<b>Efficiency Concerns</b>	<b>11</b>
<b>4</b>	<b>Some Implementation details</b>	<b>13</b>
<b>5</b>	<b>Reference</b>	<b>15</b>
	bddCacheStat . . . . .	20
	bddGbcStat . . . . .	21
	bddStat . . . . .	22
	bdd_addrf . . . . .	22
	bdd_addvarblock . . . . .	23
	bdd_intaddvarblock . . . . .	23
	bdd_allsat . . . . .	24
	bdd_and . . . . .	24
	bdd_anodecount . . . . .	25
	bdd_appall . . . . .	25
	bdd_appex . . . . .	26
	bdd_apply . . . . .	27
	bdd_appuni . . . . .	27
	bdd_autoreorder . . . . .	28
	bdd_autoreorder_times . . . . .	28
	bdd_biimp . . . . .	28
	bdd_blockfile_hook . . . . .	29
	bdd_buildcube . . . . .	29

bdd_ibuildcube	29
bdd_cachestats	30
bdd_clear_error	30
bdd_clrvarblocks	30
bdd_compose	31
bdd_constrain	31
bdd_delref	32
bdd_disable_reorder	32
bdd_done	32
bdd_enable_reorder	33
bdd_error_hook	33
bdd_errstring	34
bdd_exist	34
bdd_extvarnum	34
bdd_false	35
bdd_file_hook	35
bdd_forall	36
bdd_freepair	36
bdd_fullsatone	36
bdd_gbc_hook	37
bdd_getallocnum	37
bdd_getnodenum	38
bdd_getreorder_method	38
bdd_getreorder_times	38
bdd_high	39
bdd_imp	39
bdd_init	40
bdd_isrunning	40
bdd_ite	41
bdd_ithvar	41
bdd_level2var	42
bdd_load	42
bdd_fnload	42
bdd_low	43
bdd_makeset	43
bdd_newpair	44
bdd_nithvar	44
bdd_nodecount	45
bdd_not	45
bdd_or	45
bdd_pathcount	46
bdd_printall	46
bdd_fprintall	46
bdd_printdot	47
bdd_fprintdot	47
bdd_printorder	47
bdd_printset	48
bdd_fprintset	48
bdd_printstat	48
bdd_fprintstat	48

bdd_printtable	49
bdd_fprinttable	49
bdd_relprod	49
bdd_reorder	50
bdd_reorder_gain	51
bdd_reorder_hook	51
bdd_reorder_probe	52
bdd_reorder_verbose	52
bdd_replace	53
bdd_resetpair	53
bdd_resize_hook	54
bdd_restrict	55
bdd_satcount	55
bdd_setcountset	55
bdd_satcountln	56
bdd_setcountlnset	56
bdd_satone	56
bdd_satoneset	57
bdd_save	57
bdd_fnsave	57
bdd_scanset	58
bdd_setcacheratio	58
bdd_setmaxincrease	59
bdd_setmaxnodenum	59
bdd_setminfreenodes	60
bdd_setpair	60
bdd_setbddpair	60
bdd_setpairs	61
bdd_setbddpairs	61
bdd_setvarnum	61
bdd_setvarorder	62
bdd_simplify	62
bdd_stats	62
bdd_strm_hook	63
bdd_support	63
bdd_swapvar	64
bdd_true	64
bdd_unique	65
bdd_var	65
bdd_var2level	65
bdd_varblockall	66
bdd_varnum	66
bdd_varprofile	66
bdd_veccompose	67
bdd_versionnum	67
bdd_versionstr	67
bdd_xor	68
bddfalse	68
bddtrue	68
bvec	69

bvec_add	69
bvec_addrf	70
bvec_coerce	70
bvec_con	70
bvec_copy	71
bvec_delref	71
bvec_div	71
bvec_divfixed	72
bvec_equ	72
bvec_false	72
bvec_free	73
bvec_gte	73
bvec_gth	73
bvec_isconst	74
bvec_ite	74
bvec_lte	74
bvec_lth	75
bvec_map1	75
bvec_map2	76
bvec_map3	76
bvec_mul	77
bvec_mulfixed	77
bvec_neq	77
bvec_shl	78
bvec_shlfixed	78
bvec_shr	79
bvec_shrfixed	79
bvec_sub	80
bvec_true	80
bvec_val	81
bvec_var	81
bvec_varfdd	82
bvec_varvec	82
fdd_clearall	82
fdd_domain	83
fdd_domainnum	83
fdd_domainsize	83
fdd_equals	84
fdd_extdomain	84
fdd_file_hook	85
fdd_intaddvarblock	85
fdd_ithset	86
fdd_ithvar	86
fdd_makeset	87
fdd_overlapdomain	87
fdd_printset	88
fdd_fprintset	88
fdd_scanallvar	88
fdd_scanset	89
fdd_scanvar	89

fdd_setpair	90
fdd_setpairs	90
fdd_strm_hook	91
fdd_varnum	91
fdd_vars	92
operator<<	92





# Chapter 1

## Introduction

BuDDy is a Binary Decision Diagram package that provides all of the most used functions for manipulating BDDs. The package also includes functions for integer arithmetics such as addition and relational operators.

BuDDy started as a technology transfer project between the Technical University of Denmark and Bann Visualstate. The later is now using the techniques from BuDDy in their software. See [www.visualstate.com](http://www.visualstate.com).

This manual describes only the interface to BuDDy, not the underlying theory of BDDs. More information about that can be found in Henrik Reif Andersen's "An Introduction To Binary Decision Diagrams" which is supplied with the BuDDY distribution. Even more information can of course be found in the original papers by Bryant, Rudell and Brace [1, 3, 2, 4]

### 1.1 Acknowledgements

Thanks to the following people for new ideas, bug hunts and lots of discussions: Gerd Behrmann, Henrik Reif Andersen, Ken Larsen, Jacob Lichtenberg, Poul Williams, Nikolaj Bjorner, Alan Mishchenko, Henrik Hulgaard, and Malte Helmert.



# Chapter 2

## Users Guide

### 2.1 Getting BuDDy

BuDDy can be found on the server <http://www.itu.dk/research/buddy>.

### 2.2 Installing

1. Edit the file "config" to specify your compiler and install options.
2. Type `make` to make the binary.
3. Type `make install` to copy the BDD files to their appropriate directories
4. Type `make examples` to make the examples

### 2.3 Compiling

This is rather simple. Just inform the compiler of where the binaries and include files are installed. With Gnu C this is done with the `-I` and `-L` options. Assuming that the binary library `libbdd.a` is installed in `/usr/local/lib` and the include file `bdd.h` is installed in `/usr/local/include`, then the compile command should be

```
cc -I/usr/local/include myfile.c -o myfile -L/usr/include/lib -lbdd
```

If the above directories are included in your search path already, then you might be able to reduce the command to

```
cc myfile.c -o myfile -lbdd
```

### 2.4 Programming with BuDDy

First of all a program needs to call `bdd_init(nodenum, cachesize)` to initialize the BDD package. The `nodenum` parameter sets the initial number of BDD nodes and `cachesize` sets the size of the caches used for the BDD operators (not the unique node table). These caches are used for `bdd_apply` among others.

Good initial values are

<b>Example</b>	<b>nodenum</b>	<b>cache size</b>
Small test examples	1000	100
Small examples	10000	1000
Medium sized examples	100000	10000
Large examples	1000000	variable

Too few nodes will only result in reduced performance as this increases the number of garbage collections needed. If the package needs more nodes, then it will automatically increase the size of the node table. Use `bdd_setminfreenodes` to change the parameters for when this is done and use `bdd_setcacheratio` to enable dynamical resizing of the operator caches. You may also use the function `bdd_setmaxincrease` to adjust how BuDDy resizes the node table.

After the initialization a call must be done to `bdd_setvarnum` to define how many variables to use in this session. This number may be increased later on either by calls to `bdd_setvarnum` or to `bdd_extvarnum`.

The atomic functions for getting new BDD nodes are `bdd_ithvar(i)` and `bdd_nithvar(i)` which returns references to BDD nodes of the form  $(v_i, 0, 1)$  and  $(v_i, 1, 0)$ . The nodes constructed in this way corresponds to the positive and negative versions of a single variable. Initially the variable order is  $v_0 < v_1 < \dots < v_{n-1} < v_n$ .

The BDDs returned from `bdd_ithvar(i)` can then be used to form new BDDs by calling `bdd_apply(a,b,op)` where `op` may be `bddop_and` or any of the other operators defined in `bdd.h`. The apply function performs the binary operation indicated by `op`. Use `bdd_not` to negate a BDD. The result from `bdd_apply` and any other BDD operator *must* be handed over to `bdd_addref` to increase the reference count of the node before any other operation is performed. This is done to prevent the BDD from being garbage collected. When a BDD is no longer in use, it can be de-referenced by a call to `bdd_delref`. The exceptions to this are the return values from `bdd_ithvar` and `bdd_nithvar`. These do not need to be reference counted, although it is not an error to do so. The use of the BDD package ends with a call to `bdd_done`. See the figures 2.1 and 2.2 for an example.

Information on the BDDs can be found using the `bdd_var`, `bdd_low` and `bdd_high` functions that returns the variable labelling a BDD, the low branch and the high branch of a BDD.

Printing BDDs is done using the functions `bdd_printall` that prints *all* used nodes, `bdd_printtable` that prints the part of the nodetable that corresponds to a specific BDD and `bdd_printset` that prints a specific BDD as a list of elements in a set (all paths ending in the true terminal).

### 2.4.1 More Examples

More complex examples can be found in the `buddy/examples` directory.

## 2.5 Variable sets

For some functions like `bdd_exist` it is possible to pass a whole set of variables to be quantified, using BDDs that represent the variables. These BDDs are simply the conjunction of all the variables in their positive form and can either be build that way or by a call to `bdd_makeset`. For the `bdd_restrict` function the variables need to be included in both positive and negative form which can only be done manually.

If for example variable 1 and variable 3 are to be included in a set, then it can be done in two ways, as shown in figure 2.3.

```

#include <bdd.h>

main(void)
{
    bdd x,y,z;

    bdd_init(1000,100);
    bdd_setvarnum(5);

    x = bdd_ithvar(0);
    y = bdd_ithvar(1);
    z = bdd_addrf(bdd_apply(x,y,bddop_and));

    bdd_printtable(z);
    bdd_delref(z);
    bdd_done();
}

```

Figure 2.1: Standard C interface to BuDDy. In this mode both 'bdd' and 'BDD' can be used as BuDDy BDD types. The C interface requires the user to ensure garbage collection is handled correctly. This means calling 'bdd\_addrf' every time a new BDD is created, and 'bdd\_delref' whenever a BDD is not in use anymore.

## 2.6 Dynamic Variable Reordering

Dynamic variable reordering can be done using the functions `bdd_reorder(int method)` and `bdd_autoreorder(int method)`. Where the parameter `method`, for instance can be `BDD_REORDER_WIN2ITE`. The package must know how the BDD variables are related to each other, so the user must define blocks of BDD variables, using `bdd_addvarblock(bdd var, int fixed)`. A block is a range of BDD variables that should be kept together. It may either be a simple contiguous sequence of variables or a sequence of other blocks with ranges inside their parents range. In this way all the blocks form a tree of ranges. Partially overlapping blocks are not allowed.

Example: Assume the block  $v_0 \dots v_9$ , is added as the first block and then the block  $v_1 \dots v_8$ . This yields the  $v_0 \dots v_9$  block at the top, with the  $v_1 \dots v_8$  block as a child. If now the block  $v_1 \dots v_4$  was added, it would become a child of the  $v_1 \dots v_8$  block, similarly the block  $v_5 \dots v_8$  would be a child of the  $v_1 \dots v_8$  block. If we add the variables  $v_1$ ,  $v_2$ ,  $v_3$  and  $v_4$  as single variable blocks we at last get tree showed in figure 2.4. If all variables should be added as single variable blocks then `bdd_varblockall` can be used instead of doing it manually.

The reordering algorithm is then to first reorder the top most blocks and there after descend into each block and reorder these recursively - unless the block is defined as a fixed block.

If the user want to control the swapping of variables himself, then the functions `bdd_swapvar` `bdd_setvarorder` may be used. But this is not possible in conjunction with the use of variable blocks and the `bdd_swapvar` is unfortunately quite slow since a full scan of all the nodes must be done both before and after the swap. Other reordering functions are `bdd_autoreorder_times`, `bdd_reorder_verbose`, `bdd_sizeprobe_hook` and `bdd_reorder_hook`.

```

#include <bdd.h>

main(void)
{
    bdd x,y,z;

    bdd_init(1000,100);
    bdd_setvarnum(5);

    x = bdd_ithvar(0);
    y = bdd_ithvar(1);
    z = x & y;

    cout << bddtable << z << endl;

    bdd_done();
}

```

Figure 2.2: C++ interface to BuDDy. In this mode 'bdd' is a C++ class that wraps a handler around the standard C interface, and the 'BDD' type refers to the standard C BDD type. The C++ interface handles all garbage collection, so no calls to 'bdd\_addrf' and 'bdd\_delref' are needed.

## 2.7 Error Handling

If an error occurs then a check is done to see if there is any error handler defined and if so it is called with the error code of interest. The default error handler prints an error message on `stderr` and then aborts the program. A handler can also be defined by the user with a call to `bdd_error_hook`.

## 2.8 The C++ interface

Mostly this consists of a set of overloaded function wrappers that takes a `bdd` class and calls the appropriate C functions with the root number stored in the `bdd` class. The names of these wrappers are exactly the same as for the C functions. In addition to this a lot of the C++ operators like `|` `&` `-` `=` `==` are overloaded in order to perform most of the `bdd_apply()` operations. These are listed together with `bdd_apply`. The rest are

Operator	Description	Return value
<code>=</code>	assignment	
<code>==</code>	test	returns 1 if two BDDs are equal, otherwise 0
<code>!=</code>	test	returns 0 if two BDDs are equal, otherwise 1
<code>bdd.id()</code>	identity	returns the root number of the BDD

The default constructor for the `bdd` class initializes the bdds to the constant false value. Reference counting is totally automatic when the `bdd` class is used, here the constructors and destructors takes care of *all* reference counting! The C++ interface is also defined in `bdd.h` so nothing extra is needed to use it.

```

#include <bdd.h>

main()
{
    bdd v1, v3;
    bdd seta, setb;
    static int v[2] = {1,3};

    bdd_init(100,100);
    bdd_setvarnum(5);

    v1 = bdd_ithvar(1);
    v3 = bdd_ithvar(3);

    /* One way */
    seta = bdd_addrf( bdd_apply(v1,v3,bddop_and) );
    bdd_printtable(seta);

    /* Another way */
    setb = bdd_addrf( bdd_makeset(v,2) );
    bdd_printtable(setb);
}

```

Figure 2.3: Two ways to create a variable set.

## 2.9 Finite Domain Blocks

Included in the BDD package is a set of functions for manipulating values of finite domains, like for example finite state machines. These functions are used to allocate blocks of BDD variables to represent integer values instead of only true and false.

New finite domain blocks are allocated using `fdd_extdomain` and BDDs representing integer values can be build using `fdd_ithvar`. The BDD representing identity between two sets of different domains can be build using `fdd_equals`. BDDs representing finite domain sets can be printed using `fdd_printset` and the overloaded C++ operator `<<`. Pairs for `bdd_replace` can be made using `fdd_setpair` and variable sets can be made using `fdd_ithset` and `fdd_makeset`. The finite domain block interface is defined for both C and C++. To use this interface you must include "`fdd.h`".

Encoding using FDDs are done with the Least Significant Bits first in the ordering (top of the BDD). Assume variables  $V_0 \dots V_3$  are used to encode the value 12 - this would yield  $V_0 = 0, V_1 = 0, V_2 = 1, V_3 = 1$ .

An example program using the FDD interface can be found in the examples directory.

## 2.10 Boolean Vectors

Another interface layer for BuDDy implements boolean vectors for use with integer arithmetics. A boolean vector is simply an array of BDDs where each BDD represents one bit of an expression. To use this interface you must include "`bvec.h`". As an example, suppose we want to express the following assignment from an expression

$$x := y + 10$$

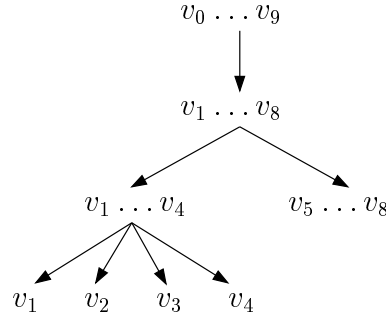


Figure 2.4: The variable tree for the variable blocks  $v_0 \dots v_9$ ,  $v_1 \dots v_8$ ,  $v_1 \dots v_4$ ,  $v_5 \dots v_8$ ,  $v_1$ ,  $v_2$ ,  $v_3$  and  $v_4$ .

what we do is to encode the variable  $y$  and the value 10 as boolean vectors  $y$  and  $v$  of a fixed length. Assume we use four bits with LSB to the right, then we get

$$y = \langle y_4, \dots, y_1 \rangle$$

$$v = \langle 1, 0, 1, 0 \rangle$$

where each  $y_i$  is the BDD variable used to encode the integer variable  $y$ . Now the result of the addition can be expressed as the vector  $z = \langle z_4, \dots, z_1 \rangle$  where each  $z_i$  is:

$$z_i = y_i \text{ xor } v_i \text{ xor } c_{i-1}$$

and the carry in  $c_i$  is

$$c_i = (y_i \text{ and } v_i) \text{ or } (c_{i-1} \text{ and } (y_i \text{ or } v_i)).$$

with  $c_0 = 0$ . What is left now is to assign the result to  $x$ . This is a conjunction of a biimplication of each element in the vectors, so the result is

$$R = \bigwedge_{i=1}^4 x_i \Leftrightarrow z_i.$$

The above example could be carried out with the following C++ program that utilizes the FDD interface for printing the result.

```

#include "bvec.h"

main()
{
    int domain[2] = {16,16};

    bdd_init(100,100);
    fdd_extdomain(domain, 2);

    bvec y = bvec_varfdd(0);
    bvec v = bvec_con(4, 10);
    bvec z = bvec_add(y, v);

    bvec x = bvec_varfdd(1);

```



```

    bdd result = bddtrue;

    for (int n=0 ; n<x.bitnum() ; n++)
        result &= bdd_apply(x[n], z[n], bddop_biimp);

    cout << fddset << result << endl << endl;
}

```

The relational operators  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $=$ ,  $\neq$  can also be encoded. Assume we want to encode  $x \leq y$  using the same variables as in the above example. This would be done as:

```

#include "bvec.h"

main()
{
    int domain[2] = {16,16};

    bdd_init(100,100);
    fdd_extdomain(domain, 2);

    bvec y = bvec_varfdd(1);
    bvec x = bvec_varfdd(0);

    bdd result = bvec_lte(x,y);

    cout << fddset << result << endl << endl;
}

```

Please note that all vectors that are returned from any of the `bvec_XXX` functions are referenced counted by the system.

### 2.10.1 C++ Interface

The C++ interface defines the class

```

class bvec
{
public:

    bvec(void);
    bvec(int bitnum);
    bvec(int bitnum, int val);
    bvec(const bvec &v);
    ~bvec(void);

    void set(int i, const bdd &b);
    bdd operator[](int i) const;
    int bitnum(void) const;
    int empty(void) const;
    bvec operator=(const bvec &src);
}

```

The default constructor makes an empty vector with no elements, the integer constructor creates a vector with `bitnum` elements (all set to false) and the third constructor creates a vector with `bitnum` elements and assigns the integer value `val` to the vector. Reference counting is done automatically. The *i*'th element in the vector can be changed with `set` and read with `operator[]`. The number of bits can be found with `bitnum` and the method `empty` returns true if the vector is a NULL vector.

## Chapter 3

# Efficiency Concerns

Getting the most out of any BDD package is not always easy. It requires some knowledge about the optimal order of the BDD variables and it also helps if you have some knowledge of the internals of the package.

First of all — a good initial variable order is a must. Using the automatic reordering methods may be an easy solution, but without a good initial order it may also be a waste of time.

Second — memory is speed. If you allocate as much memory as possible from the very beginning, then BuDDy does not have to waste time trying to allocate more whenever it is needed. So if you really want speed then `bdd_init` should be called with as many nodes as possible. This does unfortunately have the side effect that variable reordering becomes extremely slow since it has to reorder an enormous amount of nodes the first time it is triggered.

Third — the operator caches should be as big as possible. Use the function `bdd_setcacheratio` to make sure the size of these is increased whenever more nodes are allocated. *Please note that BuDDy uses a fixed number of elements for these caches as default.* You must call `bdd_setcacheratio` to change this. I have found a cache ratio of 1:64 fitting for BDDs of more than one million nodes (the solitary example). This may be a bit overkill, but it works.

Fourth — BuDDy allocates by default a maximum of 50000 nodes (1Mb RAM) every time it resizes the node table. If your problem needs millions of nodes, then this is way too small a number. Use `bdd_setmaxincrease` to increase this number. In the solitary example something like 5000000 nodes seems more reasonable.

Fifth — by default, BuDDy increases the node table whenever there is less than 20% nodes free. By increasing this value you can make BuDDy go faster and use more memory or vice versa. You can change the value with `bdd_setminfreenodes`.

So, to sum it up: if you want speed, then allocate as many nodes as possible, use small cache ratios and set `maxincrease`. If you need memory, then allocate a small number of nodes from the beginning, use a fixed size cache, do not change `maxincrease` and lower `minfreenodes`.



## Chapter 4

# Some Implementation details

- Negated pointers are not used.
- All nodes are stored in one big contiguous array which is also used as the hash table for finding identical nodes.
- The hash function used to find identical nodes from the triple (*level*, *low*, *high*) spreads all nodes evenly in the table. This means the average length of a hash chain is at most 1.
- Each node in the node table contains a reference count, the **level** of the variable (this is its position in the current variable order), the **high** and **low** part, a **hash** index used to find the first node in a hash chain and a **next** index used to link the hash chains. Each node fits into 20 bytes of memory. Other packages uses only 16 bytes for each node but in addition to this they must keep separate tables with hash table entries. The effect of this is that the total memory consumption is 20 bytes for each node on average.
- Reference counting are done on the externally referenced nodes only.
- The ANSI-C **bdd** type is an integer number referring to an index in the node table. In C++ it is a class.
- New nodes are created by doubling (or just extending) the node table, not by adding new blocks of nodes.
- Garbage collection recursively marks all nodes reachable from the externally referenced nodes before dead nodes are removed.
- Reordering interrupts the current BDD operation and restarts it again afterwards.
- Reordering changes the hash function to one where all nodes of a specific level is placed in one continuous block and updates the reference count field to include all recursive dependencies. After reordering the package returns to the normal hash function.



# Chapter 5

## Reference

Boolean vectors	
<code>bvec</code>	a boolean vector
<code>bvec_add</code>	builds a boolean vector for addition
<code>bvec_addrref</code>	increase reference count of a boolean vector
<code>bvec_coerce</code>	adjust the size of a boolean vector
<code>bvec_con</code>	build a boolean vector representing an integer value
<code>bvec_copy</code>	create a copy of a bvec
<code>bvec_delref</code>	decrease the reference count of a boolean vector
<code>bvec_div</code>	builds a boolean vector for division
<code>bvec_divfixed</code>	builds a boolean vector for division by a constant
<code>bvec_equ</code>	calculates the truth value of $x = y$
<code>bvec_false</code>	build a vector of constant false BDDs
<code>bvec_free</code>	frees all memory used by a boolean vector
<code>bvec_gte</code>	calculates the truth value of $x \geq y$
<code>bvec_gth</code>	calculates the truth value of $x > y$
<code>bvec_isconst</code>	test a vector for constant true/false BDDs
<code>bvec_ite</code>	calculates the if-then-else operator for a boolean vector
<code>bvec_lte</code>	calculates the truth value of $x \leq y$
<code>bvec_lth</code>	calculates the truth value of $x < y$
<code>bvec_map1</code>	map a function onto a boolean vector
<code>bvec_map2</code>	map a function onto a boolean vector
<code>bvec_map3</code>	map a function onto a boolean vector
<code>bvec_mul</code>	builds a boolean vector for multiplication
<code>bvec_mulfixed</code>	builds a boolean vector for multiplication with a constant
<code>bvec_neq</code>	calculates the truth value of $x \neq y$
<code>bvec_shl</code>	shift left operation (symbolic)
<code>bvec_shlfixed</code>	shift left operation (fixed number of bits)
<code>bvec_shr</code>	shift right operation (symbolic)
<code>bvec_shrfixed</code>	shift right operation
<code>bvec_sub</code>	builds a boolean vector for subtraction
<code>bvec_true</code>	build a vector of constant true BDDs
<code>bvec_val</code>	calculate the integer value represented by a boolean vector
<code>bvec_var</code>	build a boolean vector with BDD variables

bvec_varfdd	build a boolean vector from a FDD variable block
bvec_varvec	build a boolean vector with the variables passed in an array
<b>Finite domain variable blocks</b>	
fdd_clearall	clear all allocated FDD blocks
fdd_domain	bDD encoding of the domain of a FDD variable
fdd_domainnum	number of defined finite domain blocks
fdd_domainsize	real size of a finite domain block
fdd_equals	returns a BDD setting two FD. blocks equal
fdd_extdomain	adds another set of finite domain blocks
fdd_file_hook	specifies a printing callback handler
fdd_intaddvarblock	adds a new variable block for reordering
fdd_ithset	the variable set for the i'th finite domain block
fdd_ithvar	the BDD for the i'th FDD set to a specific value
fdd_makeset	creates a variable set for N finite domain blocks
fdd_overlapdomain	combine two FDD blocks into one
fdd_printset	prints a BDD for a finite domain block
fdd_fprintset	
fdd_scanallvar	finds one satisfying value of all FDD variables
fdd_scanset	scans a variable set
fdd_scanvar	finds one satisfying value of a FDD variable
fdd_setpair	defines a pair for two finite domain blocks
fdd_setpairs	defines N pairs for finite domain blocks
fdd_strm_hook	specifies a printing callback handler
fdd_varnum	binary size of a finite domain block
fdd_vars	all BDD variables associated with a finite domain block
<b>File input/output</b>	
bdd_load	loads a BDD from a file
bdd_fnload	
bdd_printall	prints all used entries in the node table
bdd_fprintall	
bdd_printdot	prints a description of a BDD in DOT format
bdd_fprintdot	
bdd_printset	prints the set of truth assignments specified by a BDD
bdd_fprintset	
bdd_printtable	prints the node table entries used by a BDD
bdd_fprinttable	
bdd_save	saves a BDD to a file
bdd_fnsave	
operator<<	c++ output operator for BDDs
<b>Information on BDDs</b>	
bdd_anodecount	counts the number of shared nodes in an array of BDDs
bdd_high	gets the true branch of a bdd
bdd_low	gets the false branch of a bdd
bdd_nodecount	counts the number of nodes used for a BDD
bdd_pathcount	count the number of paths leading to the true terminal



bdd_satcount bdd_setcountset	calculates the number of satisfying variable assignments
bdd_satcountln bdd_setcountlnset	calculates the log. number of satisfying variable assignments
bdd_support	returns the variable support of a BDD
bdd_var	gets the variable labeling the bdd
bdd_varprofile	returns a variable profile
<b>Kernel BDD operations and data structures</b>	
bddCacheStat	status information about cache usage
bddGbcStat	status information about garbage collections
bddStat	status information about the bdd package
bdd_addrf	increases the reference count on a node
bdd_cachestats	fetch cache access usage
bdd_clear_error	clears an error condition in the kernel
bdd_delref	decreases the reference count on a node
bdd_done	resets the bdd package
bdd_error_hook	set a handler for error conditions
bdd_errstring	converts an error code to a string
bdd_extvarnum	add extra BDD variables
bdd_false	returns the constant false bdd
bdd_file_hook	specifies a printing callback handler
bdd_freepair	frees a table of pairs
bdd_gbc_hook	set a handler for garbage collections
bdd_getallocnum	get the number of allocated nodes
bdd_getnodenum	get the number of active nodes in use
bdd_init	initializes the BDD package
bdd_isrunning	test whether the package is started or not
bdd_lithvar	returns a bdd representing the I'th variable
bdd_makeset	builds a BDD variable set from an integer array
bdd_newpair	creates an empty variable pair table
bdd_nithvar	returns a bdd representing the negation of the I'th variable
bdd_printstat bdd_fprintstat	print cache statistics
bdd_resetpair	clear all variable pairs
bdd_resize_hook	set a handler for nodetable resizes
bdd_scanset	returns an integer representation of a variable set
bdd_setcacheratio	sets the cache ratio for the operator caches
bdd_setmaxincrease	set max. number of nodes used to increase node table
bdd_setmaxnodenum	set the maximum available number of bdd nodes
bdd_setminfreenodes	set min. no. of nodes to be reclaimed after GBC.
bdd_setpair bdd_setbddpair	set one variable pair
bdd_setpairs bdd_setbddpairs	defines a whole set of pairs
bdd_setvarnum	set the number of used bdd variables

<code>bdd_stats</code>	returns some status information about the bdd package
<code>bdd_strm_hook</code>	specifies a printing callback handler
<code>bdd_true</code>	returns the constant true bdd
<code>bdd_varnum</code>	returns the number of defined variables
<code>bdd_versionnum</code>	returns the version number of the bdd package
<code>bdd_versionstr</code>	returns a text string with version information
<code>bddfalse</code>	the constant false bdd
<code>bddtrue</code>	the constant true bdd
<b>BDD operators</b>	
<code>bdd_allsat</code>	finds all satisfying variable assignments
<code>bdd_and</code>	the logical 'and' of two BDDs
<code>bdd_appall</code>	apply operation and universal quantification
<code>bdd_appex</code>	apply operation and existential quantification
<code>bdd_apply</code>	basic bdd operations
<code>bdd_appuni</code>	apply operation and unique quantification
<code>bdd_biimp</code>	the logical 'bi-implication' between two BDDs
<code>bdd_buildcube</code> <code>bdd_ibuildcube</code>	build a cube from an array of variables
<code>bdd_compose</code>	functional composition
<code>bdd_constrain</code>	generalized cofactor
<code>bdd_exist</code>	existential quantification of variables
<code>bdd_forall</code>	universal quantification of variables
<code>bdd_fullsatone</code>	finds one satisfying variable assignment
<code>bdd_imp</code>	the logical 'implication' between two BDDs
<code>bdd_ite</code>	if-then-else operator
<code>bdd_not</code>	negates a bdd
<code>bdd_or</code>	the logical 'or' of two BDDs
<code>bdd_relprod</code>	relational product
<code>bdd_replace</code>	replaces variables with other variables
<code>bdd_restrict</code>	restric a set of variables to constant values
<code>bdd_satone</code>	finds one satisfying variable assignment
<code>bdd_satoneset</code>	finds one satisfying variable assignment
<code>bdd_simplify</code>	coudert and Madre's restrict function
<code>bdd_unique</code>	unique quantification of variables
<code>bdd_vecompose</code>	simultaneous functional composition
<code>bdd_xor</code>	the logical 'xor' of two BDDs
<b>Variable reordering</b>	
<code>bdd_addvarblock</code> <code>bdd_intaddvarblock</code>	adds a new variable block for reordering
<code>bdd_autoreorder</code> <code>bdd_autoreorder_times</code>	enables automatic reordering
<code>bdd_blockfile_hook</code>	specifies a printing callback handler
<code>bdd_clrvarblocks</code>	clears all variable blocks
<code>bdd_disable_reorder</code>	disable automatic reordering
<code>bdd_enable_reorder</code>	enables automatic reordering

<code>bdd_getreorder_method</code>	fetch the current reorder method
<code>bdd_getreorder_times</code>	fetch the current number of allowed reorderings
<code>bdd_level2var</code>	fetch the variable number of a specific level
<code>bdd_printorder</code>	prints the current order
<code>bdd_reorder</code>	start dynamic reordering
<code>bdd_reorder_gain</code>	calculate the gain in size after a reordering
<code>bdd_reorder_hook</code>	sets a handler for automatic reorderings
<code>bdd_reorder_probe</code>	define a handler for minimization of BDDs
<code>bdd_reorder_verbose</code>	enables verbose information about reorderings
<code>bdd_setvarorder</code>	set a specific variable order
<code>bdd_swapvar</code>	swap two BDD variables
<code>bdd_var2level</code>	fetch the level of a specific BDD variable
<code>bdd_varblockall</code>	add a variable block for all variables

```
typedef struct s_bddCacheStat
{
    long unsigned int uniqueAccess;
    long unsigned int uniqueChain;
    long unsigned int uniqueHit;
    long unsigned int uniqueMiss;
    long unsigned int opHit;
    long unsigned int opMiss;
    long unsigned int swapCount;
} bddCacheStat;
```

---

**Description**

---

The fields are

<b>Name</b>	<b>Number of</b>
uniqueAccess	accesses to the unique node table
uniqueChain	iterations through the cache chains in the unique node table
uniqueHit	entries actually found in the the unique node table
uniqueMiss	entries not found in the the unique node table
opHit	entries found in the operator caches
opMiss	entries not found in the operator caches
swapCount	number of variable swaps in reordering

---

**See also**

---

bdd\_cachestats

```
typedef struct s_bddGbcStat
{
    int nodes;
    int freenodes;
    long time;
    long sumtime;
    int num;
} bddGbcStat;
```

---

**Description**

---

The fields are

<b>nodes</b>	Total number of allocated nodes in the nodetable
<b>freenodes</b>	Number of free nodes in the nodetable
<b>time</b>	Time used for garbage collection this time
<b>sumtime</b>	Total time used for garbage collection
<b>num</b>	number of garbage collections done until now

---

**See also**

---

`bdd_gbc_hook`

---

**bddStat** – Status information about the bdd package

---

```
typedef struct s_bddStat
{
    long int produced;
    int nodenum;
    int maxnodenum;
    int freenodes;
    int minfreenodes;
    int varnum;
    int cachesize;
    int gbcnum;
} bddStat;
```

---

**Description**

---

The fields are

<b>produced</b>	total number of new nodes ever produced
<b>nodenum</b>	currently allocated number of bdd nodes
<b>maxnodenum</b>	user defined maximum number of bdd nodes
<b>freenodes</b>	number of currently free nodes
<b>minfreenodes</b>	minimum number of nodes that should be left after a garbage collection.
<b>varnum</b>	number of defined bdd variables
<b>cachesize</b>	number of entries in the internal caches
<b>gbcnum</b>	number of garbage collections done until now

---

**See also**

---

bdd\_stats

---

**bdd\_addrf** – increases the reference count on a node

---

BDD bdd\_addrf(BDD r)

---

**Description**

---

Reference counting is done on externally referenced nodes only and the count for a specific node **r** can and must be increased using this function to avoid losing the node in the next garbage collection.

---

**Return value**

---

The BDD node **r**.

---

**See also**

---

bdd\_delref

---

**bdd\_addvarblock** – adds a new variable block for reordering

---

```
int bdd_addvarblock(BDD var, int fixed)
int bdd_intaddvarblock(int first, int last, int fixed)
```

### Description

---

Creates a new variable block with the variables in the variable set **var**. The variables in **var** must be contiguous. In the second form the argument **first** is the first variable included in the block and **last** is the last variable included in the block. This order does not depend on current variable order.

The variable blocks are ordered as a tree, with the largest ranges at top and the smallest at the bottom. Example: Assume the block 0-9 is added as the first block and then the block 0-6. This yields the 0-9 block at the top, with the 0-6 block as a child. If now the block 2-4 was added, it would become a child of the 0-6 block. A block of 0-8 would be a child of the 0-9 block and have the 0-6 block as a child. Partially overlapping blocks are not allowed.

The **fixed** parameter sets the block to be fixed (no reordering of its child blocks is allowed) or free, using the constants **BDD\_REORDER\_FIXED** and **BDD\_REORDER\_FREE**. Reordering is always done on the top most blocks first and then recursively downwards.

The return value is an integer that can be used to identify the block later on - with for example **bdd\_blockfile\_hook**. The values returned will be in the sequence 0, 1, 2, 3, ...

### Return value

---

A non-negative identifier on success, otherwise a negative error code.

### See also

---

**bdd\_varblockall**, **bdd\_intaddvarblock**, **bdd\_clrvarblocks**

---

**bdd\_allsat** – finds all satisfying variable assignments

---

BDD bdd\_satone(BDD r, bddallsathandler handler)

---

**Description**

---

Iterates through all legal variable assignments (those that make the BDD come true) for the bdd **r** and calls the callback handler **handler** for each of them. The array passed to **handler** contains one entry for each of the globally defined variables. Each entry is either 0 if the variable is false, 1 if it is true, and -1 if it is a don't care.

The following is an example of a callback handler that prints 'X' for don't cares, '0' for zero, and '1' for one:

```
void allsatPrintHandler(char* varset, int size)
{
    for (int v=0; v<size ; ++v)
    {
        cout << (varset[v] < 0 ? 'X' : (char)('0' + varset[v]));
    }
    cout << endl;
}
```

The handler can be used like this: `bdd_allsat(r, allsatPrintHandler);`

---

**See also**

---

`bdd_satone` `bdd_satoneset`, `bdd_fullsatone`, `bdd_satcount`, `bdd_satcountln`

---

**bdd\_and** – The logical 'and' of two BDDs

---

BDD bdd\_and(BDD l, BDD r)

---

**Description**

---

This a wrapper that calls `bdd_apply(l,r,bddop_and)`.

---

**Return value**

---

The logical 'and' of **l** and **r**.

---

**See also**

---

`bdd_apply`, `bdd_or`, `bdd_xor`



---

**bdd\_anodecount** – counts the number of shared nodes in an array of BDDs

---

```
int bdd_anodecount(BDD *r, int num)
```

---

### Description

Traverses all of the BDDs in **r** and counts all distinct nodes that are used in the BDDs—if a node is used in more than one BDD then it only counts once. The **num** parameter holds the size of the array.

---

### Return value

The number of nodes

---

### See also

bdd\_nodcount

---

**bdd\_appall** – apply operation and universal quantification

---

```
BDD bdd_appall(BDD left, BDD right, int opr, BDD var)
```

---

### Description

Applies the binary operator **opr** to the arguments **left** and **right** and then performs an universal quantification of the variables from the variable set **var**. This is done in a bottom up manner such that both the apply and quantification is done on the lower nodes before stepping up to the higher nodes. This makes the **bdd\_appall** function much more efficient than an apply operation followed by a quantification.

---

### Return value

The result of the operation.

---

### See also

bdd\_appex, bdd\_appuni, bdd\_apply, bdd\_exist, bdd\_forall, bdd\_unique, bdd\_makeset

---

**bdd\_appex** – apply operation and existential quantification

---

BDD bdd\_appex(BDD left, BDD right, int opr, BDD var)

---

**Description**

---

Applies the binary operator **opr** to the arguments **left** and **right** and then performs an existential quantification of the variables from the variable set **var**. This is done in a bottom up manner such that both the apply and quantification is done on the lower nodes before stepping up to the higher nodes. This makes the **bdd\_appex** function much more efficient than an apply operation followed by a quantification. If the operator is a conjunction then this is similar to the relational product of the two BDDs.

---

**Return value**

---

The result of the operation.

---

**See also**

---

bdd\_appall, bdd\_appuni, bdd\_apply, bdd\_exist, bdd\_forall, bdd\_unique, bdd\_makeset

---

**bdd\_apply** – basic bdd operations

---

BDD bdd\_apply(BDD left, BDD right, int opr)

---

**Description**

---

The **bdd\_apply** function performs all of the basic bdd operations with two operands, such as AND, OR etc. The **left** argument is the left bdd operand and **right** is the right operand. The **opr** argument is the requested operation and must be one of the following

Identifier	Description	Truth table	C++ opr.
bddop_and	logical and ( $A \wedge B$ )	[0,0,0,1]	<b>&amp;</b>
bddop_xor	logical xor ( $A \oplus B$ )	[0,1,1,0]	<b>^</b>
bddop_or	logical or ( $A \vee B$ )	[0,1,1,1]	<b> </b>
bddop_nand	logical not-and	[1,1,1,0]	
bddop_nor	logical not-or	[1,0,0,0]	
bddop_imp	implication ( $A \Rightarrow B$ )	[1,1,0,1]	<b>&gt;&gt;</b>
bddop_biimp	bi-implication ( $A \Leftrightarrow B$ )	[1,0,0,1]	
bddop_diff	set difference ( $A \setminus B$ )	[0,0,1,0]	<b>-</b>
bddop_less	less than ( $A < B$ )	[0,1,0,0]	<b>&lt;</b>
bddop_invimp	reverse implication ( $A \Leftarrow B$ )	[1,0,1,1]	<b>&lt;&lt;</b>

---

**Return value**

---

The result of the operation.

---

**See also**

---

bdd\_lite

---

**bdd\_appuni** – apply operation and unique quantification

---

BDD bdd\_appuni(BDD left, BDD right, int opr, BDD var)

---

**Description**

---

Applies the binary operator **opr** to the arguments **left** and **right** and then performs a unique quantification of the variables from the variable set **var**. This is done in a bottom up manner such that both the apply and quantification is done on the lower nodes before stepping up to the higher nodes. This makes the **bdd\_appuni** function much more efficient than an apply operation followed by a quantification.

---

**Return value**

---

The result of the operation.

---

**See also**

---

bdd\_appex, bdd\_appall, bdd\_apply, bdd\_exist, bdd\_unique, bdd\_forall, bdd\_makeset

---

**bdd\_autoreorder** – enables automatic reordering

---

```
int bdd_autoreorder(int method)
int bdd_autoreorder_times(int method, int num)
```

---

**Description**

---

Enables automatic reordering using `method` as the reordering method. If `method` is `BDD_REORDER_NONE` then automatic reordering is disabled. Automatic reordering is done every time the number of active nodes in the node table has been doubled and works by interrupting the current BDD operation, doing the reordering and the retrying the operation.

In the second form the argument `num` specifies the allowed number of reorderings. So if for example a "one shot" reordering is needed, then the `num` argument would be set to one.

Values for `method` can be found under `bdd_reorder`.

---

**Return value**

---

Returns the old value of `method`

---

**See also**

---

`bdd_reorder`

---

**bdd\_biimp** – The logical 'bi-implication' between two BDDs

---

```
BDD bdd_biimp(BDD l, BDD r)
```

---

**Description**

---

This a wrapper that calls `bdd_apply(l,r,bddop_biimp)`.

---

**Return value**

---

The logical 'bi-implication' of `l` and `r` ( $l \Leftrightarrow r$ ).

---

**See also**

---

`bdd_apply`, `bdd_imp`

---

**bdd\_blockfile\_hook** – Specifies a printing callback handler

---

`bddfilehandler bdd_blockfile_hook(bddfilehandler handler)`

---

### Description

---

A printing callback handler is used to convert the variable block identifiers into something readable by the end user. Use `bdd_blockfile_hook` to pass a handler to BuDDy. A typical handler could look like this:

```
void printhandler(FILE *o, int block)
{
    extern char **blocknames;
    fprintf(o, "%s", blocknames[block]);
}
```

The handler is then called from `bdd_printorder` and `bdd_reorder` (depending on the verbose level) with the block numbers returned by `bdd_addvarblock` as arguments. No default handler is supplied. The argument `handler` may be NULL if no handler is needed.

---

### Return value

---

The old handler

---

### See also

---

`bdd_printorder`

---

**bdd\_buildcube** – build a cube from an array of variables

---

`BDD bdd_buildcube(int value, int width, BDD *var)`  
`BDD bdd_ibuildcube(int value, int width, int *var)`

---

### Description

---

This function builds a cube from the variables in `var`. It does so by interpreting the `width` low order bits of `value` as a bit mask—a set bit indicates that the variable should be added in it's positive form, and a cleared bit the opposite. The most significant bits are encoded with the first variables in `var`. Consider as an example the call `bdd_buildcube(0xB, 4, var)`. This corresponds to the expression:  $var[0] \wedge \neg var[1] \wedge var[2] \wedge var[3]$ . The first version of the function takes an array of BDDs, whereas the second takes an array of variable numbers as used in `bdd_ithvar`.

---

### Return value

---

The resulting cube

---

### See also

---

`bdd_ithvar`, `fdd_ithvar`

---

**bdd\_cachestats** – Fetch cache access usage

---

```
void bdd_cachestats(bddCacheStat *s)
```

---

**Description**

---

Fetches cache usage information and stores it in **s**. The fields of **s** can be found in the documentaion for **bddCacheStat**. This function may or may not be compiled into the BuDDy package - depending on the setup at compile time of BuDDy.

---

**See also**

---

**bddCacheStat**, **bdd\_printstat**

---

**bdd\_clear\_error** – clears an error condition in the kernel

---

```
void bdd_clear_error(void)
```

---

**Description**

---

The BuDDy kernel may at some point run out of new ROBDD nodes if a maximum limit is set with **bdd\_setmaxnodenum**. In this case the current error handler is called and an internal error flag is set. Further calls to BuDDy will always return **bddfalse**. From here BuDDy must either be restarted or **bdd\_clear\_error** may be called after action is taken to let BuDDy continue. This may not be especially usefull since the default error handler exits the program - other needs may of course exist.

---

**See also**

---

**bdd\_error\_hook**, **bdd\_setmaxnodenum**

---

**bdd\_clrvarblocks** – clears all variable blocks

---

```
void bdd_clrvarblocks(void)
```

---

**Description**

---

Clears all the variable blocks that has been defined by calls to **bdd\_addvarblock**.

---

**See also**

---

**bdd\_addvarblock**

---

**bdd\_compose** – functional composition

---

BDD bdd\_compose(BDD f, BDD g, int var)

---

**Description**

---

Substitutes the variable `var` with the BDD `g` in the BDD `f`:  $\text{result} = f[g/var]$ .

---

**Return value**

---

The composed BDD

---

**See also**

---

bdd\_veccompose, bdd\_replace, bdd\_restrict

---

**bdd\_constrain** – generalized cofactor

---

BDD bdd\_constrain(BDD f, BDD c)

---

**Description**

---

Computes the generalized cofactor of `f` with respect to `c`.

---

**Return value**

---

The constrained BDD

---

**See also**

---

bdd\_restrict, bdd\_simplify

---

**bdd\_delref** – decreases the reference count on a node

---

BDD bdd\_delref(BDD r)

---

### Description

---

Reference counting is done on externally referenced nodes only and the count for a specific node **r** can and must be decreased using this function to make it possible to reclaim the node in the next garbage collection.

---

### Return value

---

The BDD node **r**.

---

### See also

---

bdd\_addrf

---

**bdd\_disable\_reorder** – Disable automatic reordering

---

void bdd\_disable\_reorder(void)

---

### Description

---

Disables automatic reordering until **bdd\_enable\_reorder** is called. Reordering is enabled by default as soon as any variable blocks have been defined.

---

### See also

---

bdd\_enable\_reorder

---

**bdd\_done** – resets the bdd package

---

void bdd\_done(void)

---

### Description

---

This function frees all memory used by the bdd package and resets the package to it's initial state.

---

### See also

---

bdd\_init



---

**bdd\_enable\_reorder** – Enables automatic reordering

---

`void bdd_enable_reorder(void)`

---

### Description

---

Re-enables reordering after a call to `bdd_disable_reorder`.

---

### See also

---

`bdd_disable_reorder`

---

**bdd\_error\_hook** – set a handler for error conditions

---

`bddinthandler bdd_error_hook(bddinthandler handler)`

---

### Description

---

Whenever an error occurs in the bdd package a test is done to see if an error handler is supplied by the user and if such exists then it will be called with an error code in the variable `errcode`. The handler may then print any usefull information and return or exit afterwards.

This function sets the handler to be `handler`. If a `NULL` argument is supplied then no calls are made when an error occurs. Possible error codes are found in `bdd.h`. The default handler is `bdd_default_errhandler` which will use `exit()` to terminate the program.

Any handler should be defined like this:

```
void my_error_handler(int errcode)
{
    ...
}
```

---

### Return value

---

The previous handler

---

### See also

---

`bdd_errstring`

---

**bdd\_errstring** – converts an error code to a string

---

`const char *bdd_errstring(int errorcode)`

---

**Description**

---

Converts a negative error code **errorcode** to a descriptive string that can be used for error handling.

---

**Return value**

---

An error description string if **e** is known, otherwise **NULL**.

---

**See also**

---

`bdd_err_hook`

---

**bdd\_exist** – existential quantification of variables

---

`BDD bdd_exist(BDD r, BDD var)`

---

**Description**

---

Removes all occurrences in **r** of variables in the set **var** by existential quantification.

---

**Return value**

---

The quantified BDD.

---

**See also**

---

`bdd_forall`, `bdd_unique`, `bdd_makeset`

---

**bdd\_extvarnum** – add extra BDD variables

---

`int bdd_extvarnum(int num)`

---

**Description**

---

Extends the current number of allocated BDD variables with **num** extra variables.

---

**Return value**

---

The old number of allocated variables or a negative error code.

---

**See also**

---

`bdd_setvarnum`, `bdd_ithvar`, `bdd_nithvar`

---

**bdd\_false** – returns the constant false bdd

---

BDD `bdd_false(void)`

---

**Description**

---

This function returns the constant false bdd and can freely be used together with the `bddtrue` and `bddfalse` constants.

---

**Return value**

---

The constant false bdd

---

**See also**

---

`bdd_true`, `bddtrue`, `bddfalse`

---

**bdd\_file\_hook** – Specifies a printing callback handler

---

`bddfilehandler bdd_file_hook(bddfilehandler handler)`

---

**Description**

---

A printing callback handler for use with BDDs is used to convert the BDD variable number into something readable by the end user. Typically the handler will print a string name instead of the number. A handler could look like this:

```
void printhandler(FILE *o, int var)
{
    extern char **names;
    fprintf(o, "%s", names[var]);
}
```

The handler can then be passed to BuDDy like this: `bdd_file_hook(printhandler)`.

No default handler is supplied. The argument `handler` may be NULL if no handler is needed.

---

**Return value**

---

The old handler

---

**See also**

---

`bdd_printset`, `bdd_strm_hook`, `fdd_file_hook`

---

**bdd\_forall** – universal quantification of variables

---

BDD bdd\_forall(BDD r, BDD var)

---

**Description**

---

Removes all occurrences in **r** of variables in the set **var** by universal quantification.

---

**Return value**

---

The quantified BDD.

---

**See also**

---

bdd\_exist, bdd\_unique, bdd\_makeset

---

**bdd\_freepair** – frees a table of pairs

---

void bdd\_freepair(bddPair \*pair)

---

**Description**

---

Frees the table of pairs **pair** that has been allocated by a call to **bdd\_newpair**.

---

**See also**

---

bdd\_replace, bdd\_newpair, bdd\_setpair, bdd\_resetpair

---

**bdd\_fullsatone** – finds one satisfying variable assignment

---

BDD bdd\_fullsatone(BDD r)

---

**Description**

---

Finds a BDD with exactly one variable at all levels. This BDD implies **r** and is not false unless **r** is false.

---

**Return value**

---

The result of the operation.

---

**See also**

---

bdd\_allsat bdd\_satone, bdd\_satoneset, bdd\_satcount, bdd\_satcountln

---

**bdd\_gbc\_hook** – set a handler for garbage collections

---

```
bddgbchandler bdd_gbc_hook(bddgbchandler handler)
```

---

**Description**

---

Whenever a garbage collection is required, a test is done to see if a handler for this event is supplied by the user and if such exists then it is called, both before and after the garbage collection takes places. This is indicated by an integer flag **pre** passed to the handler, which will be one before garbage collection and zero after garbage collection.

This function sets the handler to be **handler**. If a **NULL** argument is supplied then no calls are made when a garbage collection takes place. The argument **pre** indicates pre vs. post garbage collection and the argument **stat** contains information about the garbage collection. The default handler is **bdd\_default\_gbchandler**.

Any handler should be defined like this:

```
void my_gbc_handler(int pre, bddGbcStat *stat)
{
    ...
}
```

---

**Return value**

---

The previous handler

---

**See also**

---

**bdd\_resize\_hook**, **bdd\_reorder\_hook**

---

**bdd\_getallocnum** – get the number of allocated nodes

---

```
int bdd_getallocnum(void)
```

---

**Description**

---

Returns the number of nodes currently allocated. This includes both dead and active nodes.

---

**Return value**

---

The number of nodes.

---

**See also**

---

**bdd\_getnodenum**, **bdd\_setmaxnodenum**

---

**bdd\_getnodenum** – get the number of active nodes in use

---

`int bdd_getnodenum(void)`

---

**Description**

---

Returns the number of nodes in the nodetable that are currently in use. Note that dead nodes that have not been reclaimed yet by a garbage collection are counted as active.

---

**Return value**

---

The number of nodes.

---

**See also**

---

`bdd_getallocnum`, `bdd_setmaxnodenum`

---

**bdd\_getreorder\_method** – Fetch the current reorder method

---

`int bdd_getreorder_method(void)`

---

**Description**

---

Returns the current reorder method as defined by `bdd_autoreorder`.

---

**See also**

---

`bdd_reorder`, `bdd_getreorder_times`

---

**bdd\_getreorder\_times** – Fetch the current number of allowed reorderings

---

`int bdd_getreorder_times(void)`

---

**Description**

---

Returns the current number of allowed reorderings left. This value can be defined by `bdd_autoreorder_times`.

---

**See also**

---

`bdd_reorder_times`, `bdd_getreorder_method`

---

**bdd\_high** – gets the true branch of a bdd

---

BDD bdd\_high(BDD r)

---

**Description**

---

Gets the true branch of the bdd **r**.

---

**Return value**

---

The bdd of the true branch

---

**See also**

---

bdd\_low

---

**bdd\_imp** – The logical 'implication' between two BDDs

---

BDD bdd\_imp(BDD l, BDD r)

---

**Description**

---

This a wrapper that calls `bdd_apply(l,r,bddop_imp)`.

---

**Return value**

---

The logical 'implication' of **l** and **r** ( $l \Rightarrow r$ ).

---

**See also**

---

bdd\_apply, bdd\_biimp

---

**bdd\_init** – initializes the BDD package

---

```
int bdd_init(int nodesize, int cachesize)
```

---

**Description**

---

This function initiates the bdd package and *must* be called before any bdd operations are done. The argument **nodesize** is the initial number of nodes in the nodetable and **cachesize** is the fixed size of the internal caches. Typical values for **nodesize** are 10000 nodes for small test examples and up to 1000000 nodes for large examples. A cache size of 10000 seems to work good even for large examples, but lesser values should do it for smaller examples.

The number of cache entries can also be set to depend on the size of the nodetable using a call to **bdd\_setcacheratio**.

The initial number of nodes is not critical for any bdd operation as the table will be resized whenever there are too few nodes left after a garbage collection. But it does have some impact on the efficiency of the operations.

---

**Return value**

---

If no errors occur then 0 is returned, otherwise a negative error code.

---

**See also**

---

bdd\_done, bdd\_resize\_hook

---

**bdd\_isrunning** – test whether the package is started or not

---

```
void bdd_isrunning(void)
```

---

**Description**

---

This function tests the internal state of the package and returns a status.

---

**Return value**

---

1 (true) if the package has been started, otherwise 0.

---

**See also**

---

bdd\_init, bdd\_done



---

**bdd\_ite** – if-then-else operator

---

BDD bdd\_ite(BDD f, BDD g, BDD h)

---

**Description**

---

Calculates the BDD for the expression  $(f \wedge g) \vee (\neg f \wedge h)$  more efficiently than doing the three operations separately. **bdd\_ite** can also be used for conjunction, disjunction and any other boolean operator, but is not as efficient for the binary and unary operations.

---

**Return value**

---

The BDD for  $(f \wedge g) \vee (\neg f \wedge h)$

---

**See also**

---

bdd\_apply

---

**bdd\_ithvar** – returns a bdd representing the I'th variable

---

BDD bdd\_ithvar(int var)

---

**Description**

---

This function is used to get a bdd representing the I'th variable (one node with the childs true and false). The requested variable must be in the range define by **bdd\_setvarnum** starting with 0 being the first. For ease of use then the bdd returned from **bdd\_ithvar** does not have to be referenced counted with a call to **bdd\_addref**. The initial variable order is defined by the the index **var** that also defines the position in the variable order – variables with lower indecies are before those with higher indecies.

---

**Return value**

---

The I'th variable on succes, otherwise the constant false bdd

---

**See also**

---

bdd\_setvarnum, bdd\_nithvar, bddtrue, bddfalse

---

**bdd\_level2var** – Fetch the variable number of a specific level

---

```
int bdd_level2var(int level)
```

---

### Description

---

Returns the variable placed at position `level` in the current variable order.

---

### See also

---

bdd\_reorder, bdd\_var2level

---

**bdd\_load** – loads a BDD from a file

---

```
int bdd_fnload(char *fname, BDD *r)
int bdd_load(FILE *ifile, BDD *r)
```

---

### Description

---

Loads a BDD from a file into the BDD pointed to by `r`. The file can either be the file `ifile` which must be opened for reading or the file named `fname` which will be opened automatically for reading.

The input file format consists of integers arranged in the following manner. First the number of nodes  $N$  used by the BDD and then the number of variables  $V$  allocated and the variable ordering in use at the time the BDD was saved. If  $N$  and  $V$  are both zero then the BDD is either the constant true or false BDD, indicated by a 1 or a 0 as the next integer.

In any other case the next  $N$  sets of 4 integers will describe the nodes used by the BDD. Each set consists of first the node number, then the variable number and then the low and high nodes.

The nodes *must* be saved in a order such that any low or high node must be defined before it is mentioned.

---

### Return value

---

Zero on succes, otherwise an error code from `bdd.h`.

---

### See also

---

bdd\_save

---

**bdd\_low** – gets the false branch of a bdd

---

BDD bdd\_low(BDD r)

---

**Description**

---

Gets the false branch of the bdd **r**.

---

**Return value**

---

The bdd of the false branch

---

**See also**

---

bdd\_high

---

**bdd\_makeset** – builds a BDD variable set from an integer array

---

BDD bdd\_makeset(int \*v, int n)

---

**Description**

---

Reads a set of variable numbers from the integer array **v** which must hold exactly **n** integers and then builds a BDD representing the variable set.

The BDD variable set is represented as the conjunction of all the variables in their positive form and may just as well be made that way by the user. The user should keep a reference to the returned BDD instead of building it every time the set is needed.

---

**Return value**

---

A BDD variable set.

---

**See also**

---

bdd\_scanset

---

**bdd\_newpair** – creates an empty variable pair table

---

`bddPair *bdd_newpair(void)`

---

### Description

---

Variable pairs of the type `bddPair` are used in `bdd_replace` to define which variables to replace with other variables. This function allocates such an empty table. The table can be freed by a call to *bdd\_freepair*.

---

### Return value

---

Returns a new table of pairs.

---

### See also

---

`bdd_freepair`, `bdd_replace`, `bdd_setpair`, `bdd_setpairs`

---

**bdd\_nithvar** – returns a bdd representing the negation of the I'th variable

---

`BDD bdd_nithvar(int var)`

---

### Description

---

This function is used to get a bdd representing the negation of the I'th variable (one node with the childs false and true). The requested variable must be in the range define by `bdd_setvarnum` starting with 0 being the first. For ease of use then the bdd returned from `bdd_nithvar` does not have to be referenced counted with a call to `bdd_addrf`.

---

### Return value

---

The negated I'th variable on succes, otherwise the constant false bdd

---

### See also

---

`bdd_setvarnum`, `bdd_ithvar`, `bddtrue`, `bddfalse`

---

**bdd\_nodecount** – counts the number of nodes used for a BDD

---

int bdd\_nodecount(BDD r)

---

**Description**

---

Traverses the BDD and counts all distinct nodes that are used for the BDD.

---

**Return value**

---

The number of nodes.

---

**See also**

---

bdd\_pathcount, bdd\_satcount, bdd\_anodecount

---

**bdd\_not** – negates a bdd

---

BDD bdd\_not(BDD r)

---

**Description**

---

Negates the BDD **r** by exchanging all references to the zero-terminal with references to the one-terminal and vice versa.

---

**Return value**

---

The negated bdd.

---

**bdd\_or** – The logical 'or' of two BDDs

---

BDD bdd\_or(BDD l, BDD r)

---

**Description**

---

This a wrapper that calls `bdd_apply(1,r,bddop_or)`.

---

**Return value**

---

The logical 'or' of **l** and **r**.

---

**See also**

---

bdd\_apply, bdd\_xor, bdd\_and

---

**bdd\_pathcount** – count the number of paths leading to the true terminal

---

double bdd\_pathcount(BDD r)

---

**Description**

---

Counts the number of paths from the root node **r** leading to the terminal true node.

---

**Return value**

---

The number of paths

---

**See also**

---

bdd\_nodecount, bdd\_satcount

---

**bdd\_printall** – prints all used entries in the node table

---

void bdd\_printall(void)  
void bdd\_fprintall(FILE\* ofile)

---

**Description**

---

Prints to either stdout or the file **ofile** all the used entries in the main node table. The format is:

[Nodenum] Var/level Low High

Where **Nodenum** is the position in the node table and **level** is the position in the current variable order.

---

**See also**

---

bdd\_printtable, bdd\_printset, bdd\_printdot

---

**bdd\_printdot** – prints a description of a BDD in DOT format

---

```
void bdd_printdot(BDD r)
int bdd_fnprintdot(char* fname, BDD r)
void bdd_fprintdot(FILE* ofile, BDD r)
```

---

### Description

---

Prints a BDD in a format suitable for use with the graph drawing program DOT to either stdout, a designated file **ofile** or the file named by **fname**. In the last case the file will be opened for writing, any previous contents destroyed and then closed again.

---

### See also

---

bdd\_printall, bdd\_printtable, bdd\_printset

---

**bdd\_printorder** – prints the current order

---

```
void bdd_printorder(void)
bdd_fprint_order(FILE *f)
```

---

### Description

---

Prints an indented list of the variable blocks, showing the top most blocks to the left and the lower blocks to the right. Example:

```
2{
    0
    1
2}
3
4
```

This shows 5 variable blocks. The first one added is block zero, which is on the same level as block one. These two blocks are then sub-blocks of block two and block two is on the same level as block three and four. The numbers are the identifiers returned from **bdd\_addvarblock**. The block levels depends on the variables included in the blocks.

---

### See also

---

bdd\_reorder, bdd\_addvarblock

---

**bdd\_printset** – prints the set of truth assignments specified by a BDD

---

```
bdd_printset(BDD r)
bdd_fprintset(FILE* ofile, BDD r)
```

---

**Description**

---

Prints all the truth assignments for  $r$  that would yield it true. The format is:

```
<  $x_{1,1} : c_{1,1}, \dots, x_{1,n_1} : c_{1,n_1}$  >
<  $x_{2,1} : c_{2,1}, \dots, x_{2,n_2} : c_{2,n_2}$  >
...
<  $x_{N,1} : c_{N,1}, \dots, x_{N,n_3} : c_{N,n_3}$  >
```

Where the  $x$ 's are variable numbers (and the position in the current order) and the  $c$ 's are the possible assignments to these. Each set of brackets designates one possible assignment to the set of variables that make up the BDD. All variables not shown are don't cares. It is possible to specify a callback handler for printing of the variables using `bdd_file_hook` or `bdd_strm_hook`.

---

**See also**

---

`bdd_printall`, `bdd_printtable`, `bdd_printdot`, `bdd_file_hook`, `bdd_strm_hook`

---

**bdd\_printstat** – print cache statistics

---

```
void bdd_printstat(void)
void bdd_fprintstat(FILE *ofile)
```

---

**Description**

---

Prints information about the cache performance on standard output (or the supplied file). The information contains the number of accesses to the unique node table, the number of times a node was (not) found there and how many times a hash chain had to be traversed. Hit and miss count is also given for the operator caches.

---

**See also**

---

`bddCacheStat`, `bdd_cachestats`



---

**bdd\_printtable** – prints the node table entries used by a BDD

---

```
void bdd_printtable(BDD r)
void bdd_fprinttable(FILE* ofile, BDD r)
```

### Description

---

Prints to either stdout or the file `ofile` all the entries in the main node table used by `r`. The format is:

```
[Nodenum] Var/level :  Low High
```

Where `Nodenum` is the position in the node table and `level` is the position in the current variable order.

### See also

---

`bdd_printall`, `bdd_printset`, `bdd_printdot`

---

**bdd\_relprod** – relational product

---

```
#define bdd_relprod(a,b,var) bdd_appex(a,b,bddop_and,var)
```

### Description

---

Calculates the relational product of `a` and `b` as `a AND b` with the variables in `var` quantified out afterwards.

### Return value

---

The relational product or `bddfalse` on errors.

### See also

---

`bdd_appex`

```
void bdd_reorder(int method)
```

---

**Description**

---

This function initiates dynamic reordering using the heuristic defined by **method**, which may be one of the following

**BDD\_REORDER\_WIN2**

Reordering using a sliding window of size 2. This algorithm swaps two adjacent variable blocks and if this results in more nodes then the two blocks are swapped back again. Otherwise the result is kept in the variable order. This is then repeated for all variable blocks.

**BDD\_REORDER\_WIN2ITE**

The same as above but the process is repeated until no further progress is done. Usually a fast and efficient method.

**BDD\_REORDER\_WIN3**

The same as above but with a window size of 3.

**BDD\_REORDER\_WIN2ITE**

The same as above but with a window size of 3.

**BDD\_REORDER\_SIFT**

Reordering where each block is moved through all possible positions. The best of these is then used as the new position. Potentially a very slow but good method.

**BDD\_REORDER\_SIFTITE**

The same as above but the process is repeated until no further progress is done. Can be extremely slow.

**BDD\_REORDER\_RANDOM**

Mostly used for debugging purpose, but may be usefull for others. Selects a random position for each variable.

---

**See also**

---

`bdd_autoreorder`, `bdd_reorder_verbose`, `bdd_addvarblock`, `bdd_clrvarblocks`

---

**bdd\_reorder\_gain** – Calculate the gain in size after a reordering

---

`int bdd_reorder_gain(void)`

---

### Description

---

Returns the gain in percent of the previous number of used nodes. The value returned is

$$(100 * (A - B)) / A$$

Where  $A$  is previous number of used nodes and  $B$  is current number of used nodes.

---

**bdd\_reorder\_hook** – sets a handler for automatic reorderings

---

`bddinhandler bdd_reorder_hook(bddinhandler handler)`

---

### Description

---

Whenever automatic reordering is done, a check is done to see if the user has supplied a handler for that event. If so then it is called with the argument **prestate** being 1 if the handler is called immediately *before* reordering and **prestate** being 0 if it is called immediately after. The default handler is **bdd\_default\_reohandler** which will print information about the reordering.

A typical handler could look like this:

```
void reorderhandler(int prestate)
{
    if (prestate)
        printf("Start reordering");
    else
        printf("End reordering");
}
```

---

### Return value

---

The previous handler

---

### See also

---

`bdd_reorder`, `bdd_autoreorder`, `bdd_resize_hook`

---

**bdd\_reorder\_probe** – Define a handler for minimization of BDDs

---

`bddsizehandler bdd_reorder_probe(bddsizehandler handler)`

---

### Description

---

Reordering is typically done to minimize the global number of BDD nodes in use, but it may in some cases be usefull to minimize with respect to a specific BDD. With `bdd_reorder_probe` it is possible to define a callback function that calculates the size of a specific BDD (or anything else in fact). This handler will then be called by the reordering functions to get the current size information. A typical handle could look like this:

```
int sizehandler(void)
{
    extern BDD mybdd;
    return bdd_nodecount(mybdd);
}
```

No default handler is supplied. The argument `handler` may be NULL if no handler is needed.

---

### Return value

---

The old handler

---

### See also

---

`bdd_reorder`

---

**bdd\_reorder\_verbose** – enables verbose information about reorderings

---

`int bdd_reorder_verbose(int v)`

---

### Description

---

With `bdd_reorder_verbose` it is possible to set the level of information which should be printed during reordering. A value of zero means no information, a value of one means some information and any greater value will result in a lot of reordering information. The default value is zero.

---

### Return value

---

The old verbose level

---

### See also

---

`bdd_reorder`

---

**bdd\_replace** – replaces variables with other variables

---

BDD bdd\_replace(BDD r, bddPair \*pair)

---

**Description**

---

Replaces all variables in the BDD **r** with the variables defined by **pair**. Each entry in **pair** consists of a old and a new variable. Whenever the old variable is found in **r** then a new node with the new variable is inserted instead.

---

**Return value**

---

The result of the operation.

---

**See also**

---

bdd\_newpair, bdd\_setpair, bdd\_setpairs

---

**bdd\_resetpair** – clear all variable pairs

---

void bdd\_resetpair(bddPair \*pair)

---

**Description**

---

Resets the table of pairs **pair** by setting all substitutions to their default values (that is no change).

---

**See also**

---

bdd\_newpair, bdd\_setpair, bdd\_freepair

---

**bdd\_resize\_hook** – set a handler for nodetable resizes

---

```
bdd2inthandler bdd_resize_hook(bdd2inthandler handler)
```

---

**Description**

---

Whenever it is impossible to get enough free nodes by a garbage collection then the node table is resized and a test is done to see if a handler is supplied by the user for this event. If so then it is called with **oldsize** being the old nodetable size and **newsize** being the new nodetable size.

This function sets the handler to be **handler**. If a **NULL** argument is supplied then no calls are made when a table resize is done. No default handler is supplied.

Any handler should be defined like this:

```
void my_resize_handler(int oldsize, int newsize)
{
    ...
}
```

---

**Return value**

---

The previous handler

---

**See also**

---

`bdd_gbc_hook`, `bdd_reorder_hook`, `bdd_setminfreenodes`

---

**bdd\_restrict** – restric a set of variables to constant values

---

BDD `bdd_restrict(BDD r, BDD var)`

---

**Description**

---

This function restricts the variables in `r` to constant true or false. How this is done depends on how the variables are included in the variable set `var`. If they are included in their positive form then they are restricted to true and vice versa. Unfortunately it is not possible to insert variables in their negated form using `bdd_makeset`, so the variable set has to be build manually as a conjunction of the variables. Example: Assume variable 1 should be restricted to true and variable 3 to false.

```
bdd X = make_user_bdd();  
bdd R1 = bdd_ithvar(1);  
bdd R2 = bdd_nithvar(3);  
bdd R = bdd_addrf( bdd_apply(R1,R2, bddop_and) );  
bdd RES = bdd_addrf( bdd_restrict(X,R) );
```

---

**Return value**

---

The restricted bdd.

---

**See also**

---

`bdd_makeset`, `bdd_exist`, `bdd_forall`

---

**bdd\_satcount** – calculates the number of satisfying variable assignments

---

```
double bdd_satcount(BDD r)  
double bdd_satcountset(BDD r, BDD varset)
```

---

**Description**

---

Calculates how many possible variable assignments there exists such that `r` is satisfied (true). All defined variables are considered in the first version. In the second version, only the variables in the variable set `varset` are considered. This makes the function a *lot* slower.

---

**Return value**

---

The number of possible assignments.

---

**See also**

---

`bdd_satone`, `bdd_fullsatone`, `bdd_satcountln`

---

**bdd\_satcountln** – calculates the log. number of satisfying variable assignments

---

```
double bdd_satcountln(BDD r)
double bdd_satcountlnset(BDD r, BDD varset)
```

---

### Description

---

Calculates how many possible variable assignments there exists such that **r** is satisfied (true) and returns the logarithm of this. The result is calculated in such a manner that it is practically impossible to get an overflow, which is very possible for **bdd\_satcount** if the number of defined variables is too large. All defined variables are considered in the first version. In the second version, only the variables in the variable set **varset** are considered. This makes the function a *lot* slower!

---

### Return value

---

The logarithm of the number of possible assignments.

---

### See also

---

**bdd\_satone**, **bdd\_fullsatone**, **bdd\_satcount**

---

**bdd\_satone** – finds one satisfying variable assignment

---

```
BDD bdd_satone(BDD r)
```

---

### Description

---

Finds a BDD with at most one variable at each level. This BDD implies **r** and is not false unless **r** is false.

---

### Return value

---

The result of the operation.

---

### See also

---

**bdd\_allsat** **bdd\_satoneset**, **bdd\_fullsatone**, **bdd\_satcount**, **bdd\_satcountln**



---

**bdd\_satoneset** – finds one satisfying variable assignment

---

BDD bdd\_satoneset(BDD r, BDD var, BDD pol)

---

**Description**

---

Finds a minterm in **r**. The **var** argument is a variable set that defines a set of variables that *must* be mentioned in the result. The polarity of these variables in result—in case they are undefined in **r**—are defined by the **pol** parameter. If **pol** is the false BDD then the variables will be in negative form, and otherwise they will be in positive form.

---

**Return value**

---

The result of the operation.

---

**See also**

---

bdd\_allsat bdd\_satone, bdd\_fullsatone, bdd\_satcount, bdd\_satcountln

---

**bdd\_save** – saves a BDD to a file

---

int bdd\_fnsave(char \*fname, BDD r)  
int bdd\_save(FILE \*ofile, BDD r)

---

**Description**

---

Saves the nodes used by **r** to either a file **ofile** which must be opened for writing or to the file named **fname**. In the last case the file will be truncated and opened for writing.

---

**Return value**

---

Zero on succes, otherwise an error code from **bdd.h**.

---

**See also**

---

bdd\_load

---

**bdd\_scanset** – returns an integer representation of a variable set

---

```
int bdd_scanset(BDD r, int **v, int *n)
```

---

### Description

---

Scans a variable set **r** and copies the stored variables into an integer array of variable numbers. The argument **v** is the address of an integer pointer where the array is stored and **n** is a pointer to an integer where the number of elements are stored. It is the users responsibility to make sure the array is deallocated by a call to **free(v)**. The numbers returned are guaranteed to be in ascending order.

---

### Return value

---

Zero on success, otherwise a negative error code.

---

### See also

---

`bdd_makeset`

---

**bdd\_setcacheratio** – Sets the cache ratio for the operator caches

---

```
int bdd_setcacheratio(int r)
```

---

### Description

---

The ratio between the number of nodes in the nodetable and the number of entries in the operator cachetables is called the cache ratio. So a cache ratio of say, four, allocates one cache entry for each four unique node entries. This value can be set with **bdd\_setcacheratio** to any positive value. When this is done the caches are resized instantly to fit the new ratio. The default is a fixed cache size determined at initialization time.

---

### Return value

---

The previous cache ratio or a negative number on error.

---

### See also

---

`bdd_init`

---

**bdd\_setmaxincrease** – set max. number of nodes used to increase node table

---

```
int bdd_setmaxincrease(int size)
```

---

### Description

---

The node table is expanded by doubling the size of the table when no more free nodes can be found, but a maximum for the number of new nodes added can be set with `bdd_maxincrease` to `size` nodes. The default is 50000 nodes (1 Mb).

---

### Return value

---

The old threshold on succes, otherwise a negative error code.

---

### See also

---

`bdd_setmaxnodenum`, `bdd_setminfreenodes`

---

**bdd\_setmaxnodenum** – set the maximum available number of bdd nodes

---

```
int bdd_setmaxnodenum(int size)
```

---

### Description

---

This function sets the maximal number of bdd nodes the package may allocate before it gives up a bdd operation. The argument `size` is the absolute maximal number of nodes there may be allocated for the nodetable. Any attempt to allocate more nodes results in the constant `false` being returned and the error handler being called until some nodes are deallocated. A value of 0 is interpreted as an unlimited amount. It is *not* possible to specify fewer nodes than there has already been allocated.

---

### Return value

---

The old threshold on succes, otherwise a negative error code.

---

### See also

---

`bdd_setmaxincrease`, `bdd_setminfreenodes`

---

**bdd\_setminfreenodes** – set min. no. of nodes to be reclaimed after GBC.

---

```
int bdd_setminfreenodes(int n)
```

---

**Description**

---

Whenever a garbage collection is executed the number of free nodes left are checked to see if a resize of the node table is required. If  $X = (bddfreenum * 100) / maxnum$  is less than or equal to **n** then a resize is initiated. The range of **X** is of course 0...100 and has some influence on how fast the package is. A low number means harder attempts to avoid resizing and saves space, and a high number reduces the time used in garbage collections. The default value is 20.

---

**Return value**

---

The old threshold on succes, otherwise a negative error code.

---

**See also**

---

bdd\_setmaxnodenum, bdd\_setmaxincrease

---

**bdd\_setpair** – set one variable pair

---

```
int bdd_setpair(bddPair *pair, int oldvar, int newvar)
int bdd_setbddpair(bddPair *pair, BDD oldvar, BDD newvar)
```

---

**Description**

---

Adds the pair (**oldvar**,**newvar**) to the table of pairs **pair**. This results in **oldvar** being substituted with **newvar** in a call to **bdd\_replace**. In the first version **newvar** is an integer representing the variable to be replaced with the old variable. In the second version **oldvar** is a BDD. In this case the variable **oldvar** is substituted with the BDD **newvar**. The possibility to substitute with any BDD as **newvar** is utilized in **bdd\_compose**, whereas only the topmost variable in the BDD is used in **bdd\_replace**.

---

**Return value**

---

Zero on success, otherwise a negative error code.

---

**See also**

---

bdd\_newpair, bdd\_setpairs, bdd\_resetpair, bdd\_replace, bdd\_compose

---

**bdd\_setpairs** – defines a whole set of pairs

---

```
int bdd_setpairs(bddPair *pair, int *oldvar, int *newvar, int size)
int bdd_setbddpairs(bddPair *pair, int *oldvar, BDD *newvar, int size)
```

---

### Description

---

As for `bdd_setpair` but with `oldvar` and `newvar` being arrays of variables (BDDs) of size `size`.

---

### Return value

---

Zero on success, otherwise a negative error code.

---

### See also

---

`bdd_newpair`, `bdd_setpair`, `bdd_replace`, `bdd_compose`

---

**bdd\_setvarnum** – set the number of used bdd variables

---

```
int bdd_setvarnum(int num)
```

---

### Description

---

This function is used to define the number of variables used in the bdd package. It may be called more than one time, but only to increase the number of variables. The argument `num` is the number of variables to use.

---

### Return value

---

Zero on succes, otherwise a negative error code.

---

### See also

---

`bdd_ithvar`, `bdd_varnum`, `bdd_extvarnum`

---

**bdd\_setvarorder** – set a specific variable order

---

```
void bdd_setvarorder(int *neworder)
```

---

**Description**

---

This function sets the current variable order to be the one defined by **neworder**. The parameter **neworder** is interpreted as a sequence of variable indices and the new variable order is exactly this sequence. The array *must* contain all the variables defined so far. If for instance the current number of variables is 3 and **neworder** contains [1, 0, 2] then the new variable order is  $v_1 < v_0 < v_2$ .

---

**See also**

---

bdd\_reorder, bdd\_printorder

---

**bdd\_simplify** – coudert and Madre's restrict function

---

```
BDD bdd_simplify(BDD f, BDD d)
```

---

**Description**

---

Tries to simplify the BDD **f** by restricting it to the domain covered by **d**. No checks are done to see if the result is actually smaller than the input. This can be done by the user with a call to **bdd\_nodcount**.

---

**Return value**

---

The simplified BDD

---

**See also**

---

bdd\_restrict

---

**bdd\_stats** – returns some status information about the bdd package

---

```
void bdd_stats(bddStat* stat)
```

---

**Description**

---

This function acquires information about the internal state of the bdd package. The status information is written into the **stat** argument.

---

**See also**

---

bddStat

---

**bdd\_strm\_hook** – Specifies a printing callback handler

---

`bddstrmhandler bdd_strm_hook(bddstrmhandler handler)`

---

### Description

---

A printing callback handler for use with BDDs is used to convert the BDD variable number into something readable by the end user. Typically the handler will print a string name instead of the number. A handler could look like this:

```
void printhandler(ostream &o, int var)
{
    extern char **names;
    o << names[var];
}
```

The handler can then be passed to BuDDy like this: `bdd_strm_hook(printhandler)`.

No default handler is supplied. The argument `handler` may be NULL if no handler is needed.

---

### Return value

---

The old handler

---

### See also

---

`bdd_printset`, `bdd_file_hook`, `fdd_strm_hook`

---

**bdd\_support** – returns the variable support of a BDD

---

`BDD bdd_support(BDD r)`

---

### Description

---

Finds all the variables that `r` depends on. That is the support of `r`.

---

### Return value

---

A BDD variable set.

---

### See also

---

`bdd_makeset`

---

**bdd\_swapvar** – Swap two BDD variables

---

```
int bdd_swapvar(int v1, int v2)
```

---

**Description**

---

Use **bdd\_swapvar** to swap the position (in the current variable order) of the two BDD variables **v1** and **v2**. There are no constraints on the position of the two variables before the call. This function may *not* be used together with user defined variable blocks. The swap is done by a series of adjacent variable swaps and requires the whole node table to be rehashed twice for each call to **bdd\_swapvar**. It should therefore not be used were efficiency is a major concern.

---

**Return value**

---

Zero on succes and a negative error code otherwise.

---

**See also**

---

**bdd\_reorder**, **bdd\_addvarblock**

---

**bdd\_true** – returns the constant true bdd

---

```
BDD bdd_true(void)
```

---

**Description**

---

This function returns the constant true bdd and can freely be used together with the **bddtrue** and **bddfalse** constants.

---

**Return value**

---

The constant true bdd

---

**See also**

---

**bdd\_false**, **bddtrue**, **bddfalse**



---

**bdd\_unique** – unique quantification of variables

---

```
BDD bdd_unique(BDD r, BDD var)
```

---

**Description**

---

Removes all occurrences in **r** of variables in the set **var** by unique quantification. This type of quantification uses a XOR operator instead of an OR operator as in the existential quantification, and an AND operator as in the universal quantification.

---

**Return value**

---

The quantified BDD.

---

**See also**

---

`bdd_exist`, `bdd_forall`, `bdd_makeset`

---

**bdd\_var** – gets the variable labeling the bdd

---

```
int bdd_var(BDD r)
```

---

**Description**

---

Gets the variable labeling the bdd **r**.

---

**Return value**

---

The variable number.

---

**bdd\_var2level** – Fetch the level of a specific BDD variable

---

```
int bdd_var2level(int var)
```

---

**Description**

---

Returns the position of the variable **var** in the current variable order.

---

**See also**

---

`bdd_reorder`, `bdd_level2var`

---

**bdd\_varblockall** – add a variable block for all variables

---

```
void bdd_varblockall(void)
```

---

**Description**

---

Adds a variable block for all BDD variables declared so far. Each block contains one variable only. More variable blocks can be added later with the use of `bdd_addvarblock` – in this case the tree of variable blocks will have the blocks of single variables as the leafs.

---

**See also**

---

`bdd_addvarblock`, `bdd_intaddvarblock`

---

**bdd\_varnum** – returns the number of defined variables

---

```
int bdd_varnum(void)
```

---

**Description**

---

This function returns the number of variables defined by a call to `bdd_setvarnum`.

---

**Return value**

---

The number of defined variables

---

**See also**

---

`bdd_setvarnum`, `bdd_ithvar`

---

**bdd\_varprofile** – returns a variable profile

---

```
int *bdd_varprofile(BDD r)
```

---

**Description**

---

Counts the number of times each variable occurs in the bdd `r`. The result is stored and returned in an integer array where the `i`'th position stores the number of times the `i`'th variable occurred in the BDD. It is the users responsibility to free the array again using a call to `free`.

---

**Return value**

---

A pointer to an integer array with the profile or NULL if an error occurred.

---

**bdd\_vecompose** – simultaneous functional composition

---

BDD bdd\_vecompose(BDD f, bddPair \*pair)

---

**Description**

---

Uses the pairs of variables and BDDs in **pair** to make the simultaneous substitution:  $f[g_1/V_1, \dots, g_n/V_n]$ . In this way one or more BDDs may be substituted in one step. The BDDs in **pair** may depend on the variables they are substituting. **bdd\_compose** may be used instead of **bdd\_replace** but is not as efficient when  $g_i$  is a single variable, the same applies to **bdd\_restrict**. Note that simultaneous substitution is not necessarily the same as repeated substitution. Example:  $(x_1 \vee x_2)[x_3/x_1, x_4/x_3] = (x_3 \vee x_2) \neq ((x_1 \vee x_2)[x_3/x_1])[x_4/x_3] = (x_4 \vee x_2)$ .

---

**Return value**

---

The composed BDD

---

**See also**

---

bdd\_compose, bdd\_replace, bdd\_restrict

---

**bdd\_versionnum** – returns the version number of the bdd package

---

int bdd\_versionnum(void)

---

**Description**

---

This function returns the version number of the bdd package. The number is in the range 10-99 for version 1.0 to 9.9.

---

**See also**

---

bdd\_versionstr

---

**bdd\_versionstr** – returns a text string with version information

---

char\* bdd\_versionstr(void)

---

**Description**

---

This function returns a text string with information about the version of the bdd package.

---

**See also**

---

bdd\_versionnum

---

**bdd\_xor** – The logical 'xor' of two BDDs

---

BDD bdd\_xor(BDD l, BDD r)

---

### Description

---

This a wrapper that calls `bdd_apply(l,r,bddop_xor)`.

---

### Return value

---

The logical 'xor' of `l` and `r`.

---

### See also

---

`bdd_apply`, `bdd_or`, `bdd_and`

---

**bddfalse** – the constant false bdd

---

`extern const BDD bddfalse;`

---

### Description

---

This bdd holds the constant false value

---

### See also

---

`bddtrue`, `bdd_true`, `bdd_false`

---

**bddtrue** – the constant true bdd

---

`extern const BDD bddtrue;`

---

### Description

---

This bdd holds the constant true value

---

### See also

---

`bddfalse`, `bdd_true`, `bdd_false`

---

**bvec** – A boolean vector

---

```
typedef struct s_bvec
{
    int bitnum;
    BDD *bitvec;
} BVEC;
```

```
typedef BVEC bvec;
```

---

### Description

---

This data structure is used to store boolean vectors. The field **bitnum** is the number of elements in the vector and the field **bitvec** contains the actual BDDs in the vector. The C++ version of **bvec** is documented at the beginning of this document

---

**bvec\_add** – builds a boolean vector for addition

---

```
bvec bvec_add(bvec l, bvec r)
```

---

### Description

---

Builds a new boolean vector that represents the addition of two other vectors. Each element  $x_i$  in the result will represent the function

$$x_i = l_i \text{ xor } r_i \text{ xor } c_{i-1}$$

where the carry in  $c_i$  is

$$c_i = (l_i \text{ and } r_i) \text{ or } (c_{i-1} \text{ and } (l_i \text{ or } r_i)).$$

It is important for efficiency that the BDD variables used in **l** and **r** are interleaved.

---

### Return value

---

The result of the addition (which is already reference counted)

---

### See also

---

bvec\_sub, bvec\_mul, bvec\_shl

---

**bvec\_addrf** – increase reference count of a boolean vector

---

```
bvec bvec_addrf(bvec v)
```

---

**Description**

---

Use this function to increase the reference count of all BDDs in a `v`. Please note that all boolean vectors returned from BuDDy are reference counted from the beginning.

---

**Return value**

---

The boolean vector `v`

---

**See also**

---

`bvec_delref`

---

**bvec\_coerce** – adjust the size of a boolean vector

---

```
bvec bvec_coerce(int bitnum, bvec v)
```

---

**Description**

---

Build a boolean vector with `bitnum` elements copied from `v`. If the number of elements in `v` is greater than `bitnum` then the most significant bits are removed, otherwise if number is smaller then the vector is padded with constant false BDDs (zeros).

---

**Return value**

---

The new boolean vector (which is already reference counted)

---

**bvec\_con** – Build a boolean vector representing an integer value

---

```
bvec bvec_con(int bitnum, int val)
```

---

**Description**

---

Builds a boolean vector that represents the value `val` using `bitnum` bits. The value will be represented with the LSB at the position 0 and the MSB at position `bitnum-1`.

---

**Return value**

---

The boolean vector (which is already reference counted)

---

**See also**

---

`bvec_true`, `bvec_false`, `bvec_var`

---

**bvec\_copy** – create a copy of a bvec

---

bvec bvec\_copy(bvec src)

---

**Description**

---

Returns a copy of **src**. The result is reference counted.

---

**See also**

---

bvec\_con

---

**bvec\_delref** – decrease the reference count of a boolean vector

---

bvec bvec\_delref(bvec v)

---

**Description**

---

Use this function to decrease the reference count of all the BDDs in **v**.

---

**Return value**

---

The boolean vector **v**

---

**See also**

---

bvec\_addrf

---

**bvec\_div** – builds a boolean vector for division

---

int bvec\_div(bvec l, bvec r, bvec \*res, bvec \*rem)

---

**Description**

---

Builds a new boolean vector representing the integer division of **l** with **r**. The result of the division will be stored in **res** and the remainder of the division will be stored in **rem**. Both vectors should be initialized as the function will try to release the nodes used by them. If an error occurs then the nodes will *not* be freed.

---

**Return value**

---

Zero on success or a negative error code on error.

---

**See also**

---

bvec\_mul, bvec\_divfixed, bvec\_add, bvec\_shl

---

**bvec\_divfixed** – builds a boolean vector for division by a constant

---

```
int bvec_div(bvec e, int c, bvec *res, bvec *rem)
```

---

### Description

---

Builds a new boolean vector representing the integer division of **e** with **c**. The result of the division will be stored in **res** and the remainder of the division will be stored in **rem**. Both vectors should be initialized as the function will try to release the nodes used by them. If an error occurs then the nodes will *not* be freed.

---

### Return value

---

Zero on success or a negative error code on error.

---

### See also

---

bvec\_div, bvec\_mul, bvec\_add, bvec\_shl

---

**bvec\_equ** – calculates the truth value of  $x = y$

---

```
bdd bvec_equ(bvec l, bvec r)
```

---

### Description

---

Returns the BDD representing  $l = r$  (*not* reference counted). Both vectors must have the same number of bits.

---

### See also

---

bvec\_lth, bvec\_lte, bvec\_gth, bvec\_gte, bvec\_neq

---

**bvec\_false** – build a vector of constant false BDDs

---

```
bvec bvec_false(int bitnum)
```

---

### Description

---

Builds a boolean vector with **bitnum** elements, each of which are the constant false BDD.

---

### Return value

---

The boolean vector (which is already reference counted)

---

### See also

---

bvec\_true, bvec\_con, bvec\_var



---

**bvec\_free** – frees all memory used by a boolean vector

---

`void bvec_free(bvec v)`

---

### Description

---

Use this function to release any unused boolean vectors. The decrease of the reference counts on the BDDs in `v` is done by `bvec_free`.

---

**bvec\_gte** – calculates the truth value of  $x \geq y$

---

`bdd bvec_gte(bvec l, bvec r)`

---

### Description

---

Returns the BDD representing  $l \geq r$  (*not* reference counted). Both vectors must have the same number of bits.

---

### See also

---

`bvec_lth`, `bvec_gth`, `bvec_gth`, `bvec_equ`, `bvec_neq`

---

**bvec\_gth** – calculates the truth value of  $x > y$

---

`bdd bvec_gth(bvec l, bvec r)`

---

### Description

---

Returns the BDD representing  $l > r$  (*not* reference counted). Both vectors must have the same number of bits.

---

### See also

---

`bvec_lth`, `bvec_lte`, `bvec_gte`, `bvec_equ`, `bvec_neq`

---

**bvec\_isconst** – test a vector for constant true/false BDDs

---

```
int bvec_isconst(bvec v)
```

---

**Description**

---

Returns non-zero if the vector **v** consists of only constant true or false BDDs. Otherwise zero is returned. This test should prelude any call to **bvec\_val**.

---

**See also**

---

bvec\_val, bvec\_con

---

**bvec\_ite** – calculates the if-then-else operator for a boolean vector

---

```
bvec bvec_ite(bdd a, bvec b, bvec c)
```

---

**Description**

---

Builds a vector where the bdd **a** has been applied bitwise to **b** and **c** in an if-then-else operation, such that the result *r* is:

$$r_i = ite(a, b_i, c_i);$$

---

**Return value**

---

The if-then-else result.

---

**See also**

---

bdd\_ite

---

**bvec\_lte** – calculates the truth value of  $x \leq y$ 

---

```
bdd bvec_lte(bvec l, bvec r)
```

---

**Description**

---

Returns the BDD representing  $l \leq r$  (*not* reference counted). Both vectors must have the same number of bits.

---

**See also**

---

bvec\_lth, bvec\_gth, bvec\_gte, bvec\_equ, bvec\_neq

---

**bvec\_lth** – calculates the truth value of  $x < y$

---

`bdd bvec_lth(bvec l, bvec r)`

---

### Description

---

Returns the BDD representing  $l < r$  (*not* reference counted). Both vectors must have the same number of bits.

---

### See also

---

`bvec_lte`, `bvec_gth`, `bvec_gte`, `bvec_equ`, `bvec_neq`

---

**bvec\_map1** – map a function onto a boolean vector

---

`bvec bvec_map1(bvec a, bdd (*fun)(bdd))`

---

### Description

---

Maps the function `fun` onto all the elements in `a`. The value returned from `fun` is stored in a new vector which is then returned. An example of a mapping function is `bdd_not` which can be used like this

```
bvec res = bvec_map1(a, bdd_not)
```

to negate all the BDDs in `a`.

---

### Return value

---

The new vector (which is already reference counted)

---

### See also

---

`bvec_map2`, `bvec_map3`

---

**bvec\_map2** – map a function onto a boolean vector

---

```
bvec bvec_map2(bvec a, bvec b, bdd (*fun)(bdd,bdd))
```

---

**Description**

---

Maps the function **fun** onto all the elements in **a** and **b**. The value returned from **fun** is stored in a new vector which is then returned. An example of a mapping function is **bdd\_and** which can be used like this

```
bvec res = bvec_map2(a, b, bdd_and)
```

to calculate the logical 'and' of all the BDDs in **a** and **b**.

---

**Return value**

---

The new vector (which is already reference counted)

---

**See also**

---

**bvec\_map1**, **bvec\_map3**

---

**bvec\_map3** – map a function onto a boolean vector

---

```
bvec bvec_map3(bvec a, bvec b, bvec c, bdd (*fun)(bdd,bdd,bdd))
```

---

**Description**

---

Maps the function **fun** onto all the elements in **a**, **b** and **c**. The value returned from **fun** is stored in a new vector which is then returned. An example of a mapping function is **bdd\_ite** which can be used like this

```
bvec res = bvec_map3(a, b, c, bdd_ite)
```

to calculate the if-then-else function for each element in **a**, **b** and **c**.

---

**Return value**

---

The new vector (which is already reference counted)

---

**See also**

---

**bvec\_map1**, **bvec\_map2**

---

**bvec\_mul** – builds a boolean vector for multiplication

---

`bvec bvec_mul(bvec l, bvec r)`

---

**Description**

---

Builds a boolean vector representing the multiplication of `l` and `r`.

---

**Return value**

---

The result of the multiplication (which is already reference counted)

---

**See also**

---

`bvec_mulfixed`, `bvec_div`, `bvec_add`, `bvec_shl`

---

**bvec\_mulfixed** – builds a boolean vector for multiplication with a constant

---

`bvec bvec_mulfixed(bvec e, int c)`

---

**Description**

---

Builds a boolean vector representing the multiplication of `e` and `c`.

---

**Return value**

---

The result of the multiplication (which is already reference counted)

---

**See also**

---

`bvec_mul`, `bvec_div`, `bvec_add`, `bvec_shl`

---

**bvec\_neq** – calculates the truth value of  $x \neq y$

---

`bdd bvec_neq(bvec l, bvec r)`

---

**Description**

---

Returns the BDD representing  $l \neq r$  (*not* reference counted). Both vectors must have the same number of bits.

---

**See also**

---

`bvec_lte`, `bvec_lth`, `bvec_gth`, `bvec_gth`, `bvec_equ`

---

**bvec\_shl** – shift left operation (symbolic)

---

`bvec bvec_shl(bvec l, bvec r, BDD c)`

---

**Description**

---

Builds a boolean vector that represents `l` shifted `r` times to the left. The new empty elements will be set to `c`. The shift operation is fully symbolic and the number of bits shifted depends on the current value encoded by `r`.

---

**Return value**

---

The result of the operation (which is already reference counted)

---

**See also**

---

`bvec_add`, `bvec_mul`, `bvec_shlfixed`, `bvec_shr`

---

**bvec\_shlfixed** – shift left operation (fixed number of bits)

---

`bvec bvec_shlfixed(bvec v, int pos, BDD c)`

---

**Description**

---

Builds a boolean vector that represents `v` shifted `pos` times to the left. The new empty elements will be set to `c`.

---

**Return value**

---

The result of the operation (which is already reference counted)

---

**See also**

---

`bvec_add`, `bvec_mul`, `bvec_shl`, `bvec_shr`

---

**bvec\_shr** – shift right operation (symbolic)

---

`bvec bvec_shr(bvec l, bvec r, BDD c)`

---

**Description**

---

Builds a boolean vector that represents `l` shifted `r` times to the right. The new empty elements will be set to `c`. The shift operation is fully symbolic and the number of bits shifted depends on the current value encoded by `r`.

---

**Return value**

---

The result of the operation (which is already reference counted)

---

**See also**

---

`bvec_add`, `bvec_mul`, `bvec_shl`, `bvec_shrfixed`

---

**bvec\_shrfixed** – shift right operation

---

`bvec bvec_shrfixed(bvec v, int pos, BDD c)`

---

**Description**

---

Builds a boolean vector that represents `v` shifted `pos` times to the right. The new empty elements will be set to `c`.

---

**Return value**

---

The result of the operation (which is already reference counted)

---

**See also**

---

`bvec_add`, `bvec_mul`, `bvec_shr`, `bvec_shl`

---

**bvec\_sub** – builds a boolean vector for subtraction

---

```
bvec bvec_sub(bvec l, bvec r)
```

---

**Description**

---

Builds a new boolean vector that represents the subtraction of two other vectors. Each element  $x_i$  in the result will represent the function

$$x_i = l_i \text{ xor } r_i \text{ xor } c_{i-1}$$

where the carry in  $c_i$  is

$$c_i = (l_i \text{ and } r_i \text{ and } c_{i-1}) \text{ or } (\text{not } l_i \text{ and } (r_i \text{ or } c_{i-1})).$$

It is important for efficiency that the BDD variables used in **l** and **r** are interleaved.

---

**Return value**

---

The result of the subtraction (which is already reference counted)

---

**See also**

---

bvec\_add, bvec\_mul, bvec\_shl

---

**bvec\_true** – build a vector of constant true BDDs

---

```
bvec bvec_true(int bitnum)
```

---

**Description**

---

Builds a boolean vector with **bitnum** elements, each of which are the constant true BDD.

---

**Return value**

---

The boolean vector (which is already reference counted)

---

**See also**

---

bvec\_false, bvec\_con, bvec\_var



---

**bvec\_val** – calculate the integer value represented by a boolean vector

---

```
int bvec_val(bvec v)
```

---

### Description

---

Calculates the value represented by the bits in **v** assuming that the vector **v** consists of only constant true or false BDDs. The LSB is assumed to be at position zero.

---

### Return value

---

The integer value represented by **v**.

---

### See also

---

bvec\_isconst, bvec\_con

---

**bvec\_var** – build a boolean vector with BDD variables

---

```
bvec bvec_var(int bitnum, int offset, int step)
```

---

### Description

---

Builds a boolean vector with the BDD variables  $v_1, \dots, v_n$  as the elements. Each variable will be the the variabed numbered **offset + N\*step** where **N** ranges from 0 to **bitnum-1**.

---

### Return value

---

The boolean vector (which is already reference counted)

---

### See also

---

bvec\_true, bvec\_false, bvec\_con

---

**bvec\_varfdd** – build a boolean vector from a FDD variable block

---

```
bvec bvec_varfdd(int var)
```

---

### Description

Builds a boolean vector which will include exactly the variables used to define the FDD variable block **var**. The vector will have the LSB at position zero.

---

### Return value

The boolean vector (which is already reference counted)

---

### See also

bvec\_var

---

**bvec\_varvec** – build a boolean vector with the variables passed in an array

---

```
bvec bvec_varvec(int bitnum, int *var)
```

---

### Description

Builds a boolean vector with the BDD variables listed in the array **var**. The array must be of size **bitnum**.

---

### Return value

The boolean vector (which is already reference counted)

---

### See also

bvec\_var

---

**fdd\_clearall** – clear all allocated FDD blocks

---

```
void fdd_clearall(void)
```

---

### Description

Removes all defined finite domain blocks defined by **fdd\_extdomain()** and **fdd\_overlapdomain()**

---

**fdd\_domain** – BDD encoding of the domain of a FDD variable

---

BDD `fdd_domain(int var)`

---

**Description**

---

Returns what corresponds to a disjunction of all possible values of the variable `var`. This is more efficient than doing `fdd_ithvar(var,0)` OR `fdd_ithvar(var,1)` ... explicitly for all values in the domain of `var`.

---

**Return value**

---

The encoding of the domain

---

**fdd\_domainnum** – number of defined finite domain blocks

---

int `fdd_domainnum(void)`

---

**Description**

---

Returns the number of finite domain blocks define by calls to `bdd_extdomain`.

---

**Return value**

---

The number of defined finite domain blocks or a negative error code

---

**See also**

---

`fdd_domainsize`, `fdd_extdomain`

---

**fdd\_domainsize** – real size of a finite domain block

---

int `fdd_domainsize(int var)`

---

**Description**

---

Returns the size of the domain for the finite domain block `var`.

---

**Return value**

---

The size or a negative error code

---

**See also**

---

`fdd_domainnum`

---

**fdd\_equals** – returns a BDD setting two FD. blocks equal

---

BDD fdd\_equals(int f, int g)

---

### Description

---

Builds a BDD which is true for all the possible assignments to the variable blocks **f** and **g** that makes the blocks equal. This is more or less just a shorthand for calling **fdd\_equ()**.

---

### Return value

---

The correct BDD or the constant false on errors.

---

**fdd\_extdomain** – adds another set of finite domain blocks

---

int fdd\_extdomain(int \*dom, int num)

---

### Description

---

Extends the set of finite domain blocks with the **num** domains in **dom**. Each entry in **dom** defines the size of a new finite domain which later on can be used for finite state machine traversal and other operations on finite domains. Each domain allocates  $\log_2(|dom[i]|)$  BDD variables to be used later. The ordering is interleaved for the domains defined in each call to **fdd\_extdomain**. This means that assuming domain  $D_0$  needs 2 BDD variables  $x_1$  and  $x_2$ , and another domain  $D_1$  needs 4 BDD variables  $y_1, y_2, y_3$  and  $y_4$ , then the order will be  $x_1, y_1, x_2, y_2, y_3, y_4$ . The index of the first domain in **dom** is returned. The index of the other domains are offset from this index with the same offset as in **dom**.

The BDD variables needed to encode the domain are created for the purpose and do not interfere with the BDD variables already in use.

---

### Return value

---

The index of the first domain or a negative error code.

---

### See also

---

fdd\_lithvar, fdd\_equals, fdd\_overlapdomain

---

**fdd\_file\_hook** – Specifies a printing callback handler

---

`bddfilehandler fdd_file_hook(bddfilehandler handler)`

---

### Description

---

A printing callback handler for use with FDDs is used to convert the FDD integer identifier into something readable by the end user. Typically the handler will print a string name instead of the identifier. A handler could look like this:

```
void printhandler(FILE *o, int var)
{
    extern char **names;
    fprintf(o, "%s", names[var]);
}
```

The handler can then be passed to BuDDy like this: `fdd_file_hook(printhandler)`.

No default handler is supplied. The argument `handler` may be `NULL` if no handler is needed.

---

### Return value

---

The old handler

---

### See also

---

`fdd_printset`, `bdd_file_hook`

---

**fdd\_intaddvarblock** – adds a new variable block for reordering

---

`int fdd_intaddvarblock(int first, int last, int fixed)`

---

### Description

---

Works exactly like `bdd_addvarblock` except that `fdd_intaddvarblock` takes a range of FDD variables instead of BDD variables.

---

### Return value

---

Zero on success, otherwise a negative error code.

---

### See also

---

`bdd_addvarblock`, `bdd_intaddvarblock`, `bdd_reorder`

---

**fdd\_ithset** – the variable set for the i'th finite domain block

---

BDD fdd\_ithset(int var)

---

**Description**

---

Returns the variable set that contains the variables used to define the finite domain block **var**.

---

**Return value**

---

The variable set or the constant false BDD on error.

---

**See also**

---

fdd\_ithvar

---

**fdd\_ithvar** – the BDD for the i'th FDD set to a specific value

---

BDD fdd\_ithvar(int var, int val)

---

**Description**

---

Returns the BDD that defines the value **val** for the finite domain block **var**. The encoding places the Least Significant Bit at the top of the BDD tree (which means they will have the lowest variable index). The returned BDD will be  $V_0 \wedge V_1 \wedge \dots \wedge V_N$  where each  $V_i$  will be in positive or negative form depending on the value of **val**.

---

**Return value**

---

The correct BDD or the constant false BDD on error.

---

**See also**

---

fdd\_ithset

---

**fdd\_makeset** – creates a variable set for N finite domain blocks

---

BDD fdd\_makeset(int \*varset, int varnum)

---

### Description

---

Returns a BDD defining all the variable sets used to define the variable blocks in the array **varset**. The argument **varnum** defines the size of **varset**.

---

### Return value

---

The correct BDD or the constant false on errors.

---

### See also

---

fdd\_lthset, bdd\_makeset

---

**fdd\_overlapdomain** – combine two FDD blocks into one

---

int fdd\_overlapdomain(int v1, int v2)

---

### Description

---

This function takes two FDD blocks and merges them into a new one, such that the new one is encoded using both sets of BDD variables. If **v1** is encoded using the BDD variables  $a_1, \dots, a_n$  and has a domain of  $[0, N_1]$ , and **v2** is encoded using  $b_1, \dots, b_n$  and has a domain of  $[0, N_2]$ , then the result will be encoded using the BDD variables  $a_1, \dots, a_n, b_1, \dots, b_n$  and have the domain  $[0, N_1 * N_2]$ . The use of this function may result in some strange output from **fdd\_printset**.

---

### Return value

---

The index of the finite domain block

---

### See also

---

fdd\_extdomain

---

**fdd\_printset** – prints a BDD for a finite domain block

---

```
void fdd_printset(BDD r)
void fdd_fprintset(FILE *ofile, BDD f)
```

---

### Description

---

Prints the BDD **f** using a set notation as in **bdd\_printset** but with the index of the finite domain blocks included instead of the BDD variables. It is possible to specify a printing callback function with **fdd\_file\_hook** or **fdd\_strm\_hook** which can be used to print the FDD identifier in a readable form.

---

### See also

---

**bdd\_printset**, **fdd\_file\_hook**, **fdd\_strm\_hook**

---

**fdd\_scanallvar** – Finds one satisfying value of all FDD variables

---

```
int* fdd_scanallvar(BDD r)
```

---

### Description

---

Finds one satisfying assignment in **r** of all the defined FDD variables. Each value is stored in an array which is returned. The size of this array is exactly the number of FDD variables defined. It is the user's responsibility to free this array using **free()**.

---

### Return value

---

An array with all satisfying values. If **r** is the trivially false BDD, then NULL is returned.

---

### See also

---

**fdd\_scanvar**



---

**fdd\_scanset** – scans a variable set

---

```
int fdd_scanset(BDD r, int **varset, int *varnum)
```

---

**Description**

---

Scans the BDD **r** to find all occurrences of FDD variables and then stores these in **varset**. **varset** will be set to point to an array of size **varnum** which will contain the indices of the found FDD variables. It is the users responsibility to free **varset** after use.

---

**Return value**

---

Zero on success or a negative error code on error.

---

**See also**

---

fdd\_makeset

---

**fdd\_scanvar** – Finds one satisfying value of a FDD variable

---

```
int fdd_scanvar(BDD r, int var)
```

---

**Description**

---

Finds one satisfying assignment of the FDD variable **var** in the BDD **r** and returns this value.

---

**Return value**

---

The value of a satisfying assignment of **var**. If **r** is the trivially false BDD, then a negative value is returned.

---

**See also**

---

fdd\_scanallvar

---

**fdd\_setpair** – defines a pair for two finite domain blocks

---

```
int fdd_setpair(bddPair *pair, int p1, int p2)
```

---

**Description**

---

Defines each variable in the finite domain block **p1** to be paired with the corresponding variable in **p2**. The result is stored in **pair** which must be allocated using **bdd\_makepair**.

---

**Return value**

---

Zero on success or a negative error code on error.

---

**See also**

---

**fdd\_setpairs**

---

**fdd\_setpairs** – defines N pairs for finite domain blocks

---

```
int fdd_setpairs(bddPair *pair, int *p1, int *p2, int size)
```

---

**Description**

---

Defines each variable in all the finite domain blocks listed in the array **p1** to be paired with the corresponding variable in **p2**. The result is stored in **pair** which must be allocated using **bdd\_makeset**.

---

**Return value**

---

Zero on success or a negative error code on error.

---

**See also**

---

**bdd\_setpair**

---

**fdd\_strm\_hook** – Specifies a printing callback handler

---

`bddstrmhandler fdd_strm_hook(bddstrmhandler handler)`

---

### Description

---

A printing callback handler for use with FDDs is used to convert the FDD integer identifier into something readable by the end user. Typically the handler will print a string name instead of the identifier. A handler could look like this:

```
void printhandler(ostream &o, int var)
{
    extern char **names;
    o << names[var];
}
```

The handler can then be passed to BuDDy like this: `fdd_strm_hook(printhandler)`.

No default handler is supplied. The argument `handler` may be NULL if no handler is needed.

---

### Return value

---

The old handler

---

### See also

---

`fdd_printset`, `bdd_file_hook`

---

**fdd\_varnum** – binary size of a finite domain block

---

`int fdd_varnum(int var)`

---

### Description

---

Returns the number of BDD variables used for the finite domain block `var`.

---

### Return value

---

The number of variables or a negative error code

---

### See also

---

`fdd_vars`

---

**fdd\_vars** – all BDD variables associated with a finite domain block

---

```
int *fdd_vars(int var)
```

---

### Description

---

Returns an integer array containing the BDD variables used to define the finite domain block `var`. The size of the array is the number of variables used to define the finite domain block. The array will have the Least Significant Bit at pos 0. The array must *not* be deallocated.

---

### Return value

---

Integer array containing the variable numbers or NULL if `v` is an unknown block.

---

### See also

---

fdd\_varnum

---

**operator<<** – C++ output operator for BDDs

---

```
ostream &operator<<(ostream &o, const bdd_ioformat &f)
ostream &operator<<(ostream &o, const bdd &r)
```

---

### Description

---

BDDs can be printed in various formats using the C++ iostreams library. The formats are the those used in `bdd_printset`, `bdd_printtable`, `fdd_printset` and `bdd_printdot`. The format can be specified with the following format objects:

<code>bddset</code>	BDD level set format
<code>bddtable</code>	BDD level table format
<code>bdddot</code>	Output for use with Dot
<code>bddall</code>	The whole node table
<code>fddset</code>	FDD level set format

So a BDD `x` can for example be printed as a table with the command

```
cout << bddtable << x << endl.
```

---

### Return value

---

The specified output stream

---

### See also

---

`bdd_strm_hook`, `fdd_strm_hook`

# Bibliography

- [1] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [2] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [3] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.
- [4] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *IEEE/ACM International Conference on CAD*, pages 42–47, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

# Index

addition, 8  
arithmetic, 7

bdd\_addrf, 22  
bdd\_addvarblock, 23  
bdd\_allsat, 24  
bdd\_and, 24  
bdd\_anodecount, 25  
bdd\_appall, 25  
bdd\_appex, 26  
bdd\_apply, 27  
bdd\_appuni, 27  
bdd\_autoreorder, 28  
bdd\_autoreorder\_times, 28  
bdd\_biimp, 28  
bdd\_blockfile\_hook, 29  
bdd\_buildcube, 29  
bdd\_cachestats, 30  
bdd\_clear\_error, 30  
bdd\_clrvarblocks, 30  
bdd\_compose, 31  
bdd\_constrain, 31  
bdd\_delref, 32  
bdd\_disable\_reorder, 32  
bdd\_done, 32  
bdd\_enable\_reorder, 33  
bdd\_error\_hook, 33  
bdd\_errstring, 34  
bdd\_exist, 34  
bdd\_extvarnum, 34  
bdd\_false, 35  
bdd\_file\_hook, 35  
bdd\_fnload, 42  
bdd\_fnsave, 57  
bdd\_forall, 36  
bdd\_fprintall, 46  
bdd\_fprintdot, 47  
bdd\_fprintset, 48  
bdd\_fprintstat, 48  
bdd\_fprinttable, 49  
bdd\_freepair, 36  
bdd\_fullsatone, 36  
bdd\_gbc\_hook, 37  
bdd\_getallocnum, 37  
bdd\_getnodenum, 38  
bdd\_getreorder\_method, 38  
bdd\_getreorder\_times, 38  
bdd\_high, 39  
bdd\_ibuildcube, 29  
bdd\_imp, 39  
bdd\_init, 40  
bdd\_intaddvarblock, 23  
bdd\_isrunning, 40  
bdd\_ite, 41  
bdd\_ithvar, 41  
bdd\_level2var, 42  
bdd\_load, 42  
bdd\_low, 43  
bdd\_makeset, 43  
bdd\_newpair, 44  
bdd\_nithvar, 44  
bdd\_nodecount, 45  
bdd\_not, 45  
bdd\_or, 45  
bdd\_pathcount, 46  
bdd\_printall, 46  
bdd\_printdot, 47  
bdd\_printorder, 47  
bdd\_printset, 48  
bdd\_printstat, 48  
bdd\_printtable, 49  
bdd\_relprod, 49  
bdd\_reorder, 50  
bdd\_reorder\_gain, 51  
bdd\_reorder\_hook, 51  
bdd\_reorder\_probe, 52  
bdd\_reorder\_verbose, 52  
bdd\_replace, 53  
bdd\_resetpair, 53  
bdd\_resize\_hook, 54  
bdd\_restrict, 55  
bdd\_satcount, 55  
bdd\_satcountln, 56  
bdd\_satone, 56

- bdd\_satoneset , 57
- bdd\_save , 57
- bdd\_scanset , 58
- bdd\_setbddpair , 60
- bdd\_setbddpairs , 61
- bdd\_setcacheratio , 58
- bdd\_setcountlnset , 56
- bdd\_setcountset , 55
- bdd\_setmaxincrease , 59
- bdd\_setmaxnodenum , 59
- bdd\_setminfreenodes , 60
- bdd\_setpair , 60
- bdd\_setpairs , 61
- bdd\_setvarnum , 61
- bdd\_setvarorder , 62
- bdd\_simplify , 62
- bdd\_stats , 62
- bdd\_strm\_hook , 63
- bdd\_support , 63
- bdd\_swapvar , 64
- bdd\_true , 64
- bdd\_unique , 65
- bdd\_var , 65
- bdd\_var2level , 65
- bdd\_varblockall , 66
- bdd\_varnum , 66
- bdd\_varprofile , 66
- bdd\_veccompose , 67
- bdd\_versionnum , 67
- bdd\_versionstr , 67
- bdd\_xor , 68
- bddCacheStat , 20
- bddfalse, 68
- bddGbcStat , 21
- bddStat , 22
- bddtrue , 68
- Boolean Vectors, 7
- bvec , 69
- bvec\_add , 69
- bvec\_addrref , 70
- bvec\_coerce , 70
- bvec\_con , 70
- bvec\_copy , 71
- bvec\_delref , 71
- bvec\_div , 71
- bvec\_divfixed , 72
- bvec\_equ , 72
- bvec\_false , 72
- bvec\_free , 73
- bvec\_gte , 73
- bvec\_gth , 73
- bvec\_isconst , 74
- bvec\_ite , 74
- bvec\_lte , 74
- bvec\_lth , 75
- bvec\_map1 , 75
- bvec\_map2 , 76
- bvec\_map3 , 76
- bvec\_mul , 77
- bvec\_mulfixed , 77
- bvec\_neq , 77
- bvec\_shl , 78
- bvec\_shlfixed , 78
- bvec\_shr , 79
- bvec\_shrfixed , 79
- bvec\_sub , 80
- bvec\_true , 80
- bvec\_val , 81
- bvec\_var , 81
- bvec\_varfdd , 82
- bvec\_varvec , 82
- C++ interface, 6
- compiling, 3
- dynamic variable reordering, 5
- error handling, 6
- fdd\_clearall , 82
- fdd\_domain , 83
- fdd\_domainnum , 83
- fdd\_domainsize , 83
- fdd\_equals , 84
- fdd\_extdomain , 84
- fdd\_file\_hook , 85
- fdd\_fprintset , 88
- fdd\_intaddvarblock , 85
- fdd\_ithset , 86
- fdd\_ithvar , 86
- fdd\_makeset , 87
- fdd\_overlapdomain , 87
- fdd\_printset , 88
- fdd\_scanallvar , 88
- fdd\_scanset , 89
- fdd\_scanvar , 89
- fdd\_setpair , 90
- fdd\_setpairs , 90
- fdd\_strm\_hook , 91
- fdd\_varnum , 91
- fdd\_vars , 92

finite domain blocks, 7

implementation, 13

installing, 3

operator $\ll$ , 92

programming examples, 3

relational product, 26

reordering, 5

variable reordering, 5

variable sets, 4