

Xpress-BCL

Reference manual

Release 4.0

Published by Fair Isaac Corporation

©Copyright Fair Isaac Corporation 2008. All rights reserved.

All trademarks referenced in this manual that are not the property of Fair Isaac are acknowledged.

All companies, products, names and data contained within this book are completely fictitious and are used solely to illustrate the use of Xpress-MP. Any similarity between these names or data and reality is purely coincidental.

How to Contact Fair Isaac

USA, Canada and all Americas

Information and Sales: info@dashoptimization.com

Licensing: license-usa@dashoptimization.com

Product Support: support-usa@dashoptimization.com

Tel: +1 (201) 567 9445

Fax: +1 (201) 567 9443

Fair Isaac
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Japan

Information and Sales: info@jp.dashoptimization.com

Licensing: license@jp.dashoptimization.com

Product Support: support@jp.dashoptimization.com

Tel: +81 43 297 8836

Fax: +81 43 297 8827

Dash Optimization Japan
WBG Marive-East 21F FASuC B2124
2-6 Nakase Mihama-ku
261-7121 Chiba
Japan

Worldwide

Information and Sales: info@dashoptimization.com

Licensing: license@dashoptimization.com

Product Support: support@dashoptimization.com

Tel: +44 1926 315862

Fax: +44 1926 315854

Fair Isaac
Leam House, 64 Trinity Street
Leamington Spa
Warwickshire CV32 5YN
UK

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at <http://www.dashoptimization.com> or subscribe to our mailing list.

Contents

1	Introduction	1
1.1	An overview of Xpress-BCL	1
1.2	Note for Optimizer library users	1
1.3	Structure of this manual	2
1.4	Conventions used	2
I	Modeling with BCL	3
2	Modeling with BCL	4
2.1	Problem handling	4
2.1.1	Initialization and termination	4
2.1.2	Problem creation and deletion	4
2.1.3	Other basic functions	5
2.1.4	Input and output settings	5
2.1.5	Error handling	5
2.2	Variables	6
2.2.1	Basic functions	6
2.2.2	Variable arrays	6
2.3	Constraints	7
2.3.1	Basic functions	7
2.3.2	Predefined constraint functions	8
2.3.3	Objective function	8
2.4	Solving and solution information	9
2.5	Example	10
2.5.1	Model formulation using basic functions	10
2.5.2	Using variable arrays	11
2.5.3	Completing the example: problem solving and output	12
3	Further modeling topics	14
3.1	Data input and index sets	14
3.1.1	Example	14
3.2	Special Ordered Sets	16
3.2.1	Basic functions	16
3.2.2	Array-based SOS definition	17
3.2.3	Example	17
3.3	Output and printing	18
3.3.1	Example	18
3.4	Quadratic Programming with BCL	19
3.4.1	Example	20
3.5	User error handling	21
II	BCL library and class reference	24
4	BCL C library functions	25
4.1	Layout for function descriptions	25
	XPRBaddarrterm	26

XPRBaddcutarrterm	27
XPRBaddcuts	28
XPRBaddcutterm	29
XPRBaddidxel	30
XPRBaddqterm	31
XPRBaddsosarrel	32
XPRBaddsosel	33
XPRBaddterm	34
XPRBapparrvarel	35
XPRBclearidir	36
XPRBdefcbdelvar	37
XPRBdefcberr	38
XPRBdefcbmsg	39
XPRBdelarrvar	40
XPRBdelbasis	41
XPRBdelctr	42
XPRBdelcut	43
XPRBdelcutterm	44
XPRBdelprob	45
XPRBdelqterm	46
XPRBdelsos	47
XPRBdelsosel	48
XPRBdelterm	49
XPRBendarrvar	50
XPRBexportprob	51
XPRBfinish, XPRBfree	52
XPRBfixvar	53
XPRBgetact	54
XPRBgetarrvarname	55
XPRBgetarrvarsize	56
XPRBgetbounds	57
XPRBgetbyname	58
XPRBgetcolnum	59
XPRBgetctrname	60
XPRBgetctrrng	61
XPRBgetctrtype	62
XPRBgetcutid	63
XPRBgetcutrhs	64
XPRBgetcuttype	65
XPRBgetdual	66
XPRBgetidxel	67
XPRBgetidxelname	68
XPRBgetidxsetname	69
XPRBgetidxsetsize	70
XPRBgetiis	71
XPRBgetlim	72
XPRBgetlpstat	73
XPRBgetmipstat	74
XPRBgetmodcut	75
XPRBgetnumiis	76
XPRBgetobjval	77
XPRBgetprobname	78
XPRBgetprobstat	79
XPRBgetrange	80
XPRBgetrcost	81
XPRBgetrhs	82
XPRBgetrownum	83
XPRBgetsense	84

XPRBgetslack	85
XPRBgetsol	86
XPRBgetsosname	87
XPRBgetsostype	88
XPRBgettime	89
XPRBgetvarlink	90
XPRBgetvarname	91
XPRBgetvarrng	92
XPRBgetvartype	93
XPRBgetversion	94
XPRBgetXPRSProb	95
XPRBinit	96
XPRBloadbasis	97
XPRBloadmat	98
XPRBmaxim	99
XPRBminim	100
XPRBnewarrsum	101
XPRBnewarrvar	102
XPRBnewctr	103
XPRBnewcut	104
XPRBnewcutarrsum	105
XPRBnewcutprec	106
XPRBnewcutsum	107
XPRBnewidxset	108
XPRBnewname	109
XPRBnewprec	110
XPRBnewprob	111
XPRBnewsos	112
XPRBnewsosrc	113
XPRBnewsosw	114
XPRBnewsum	115
XPRBnewvar	116
XPRBprintarrvar	117
XPRBprintctr	118
XPRBprintcut	119
XPRBprintf	120
XPRBprintidxset	121
XPRBprintobj	122
XPRBprintprob	123
XPRBprintsos	124
XPRBprintvar	125
XPRBreadarrlinecb	126
XPRBreadlinecb	127
XPRBresetprob	128
XPRBsavebasis	129
XPRBsetarrvarel	130
XPRBsetcolororder	131
XPRBsetctrtype	132
XPRBsetcutid	133
XPRBsetcutmode	134
XPRBsetcutterm	135
XPRBsetcuttype	136
XPRBsetdecsign	137
XPRBsetdictionarysize	138
XPRBseterrctrl	139
XPRBsetlb	140
XPRBsetlim	141
XPRBsetmodcut	142

XPRBsetmsglevel	143
XPRBsetobj	144
XPRBsetqterm	145
XPRBsetrange	146
XPRBsetrealfmt	147
XPRBsetsense	148
XPRBsetosdir	149
XPRBsetterm	150
XPRBsetub	151
XPRBsetvardir	152
XPRBsetvarlink	153
XPRBsetvartype	154
XPRBsolve	155
XPRBstartarrvar	156
XPRBsync	157
XPRBwritedir	158
5 BCL in C++	159
5.1 An overview of BCL in C++	159
5.1.1 Example	159
5.1.2 QCQP Example	163
5.2 C++ class reference	163
XPRB	165
getTime	165
getVersion	165
init	166
setColOrder	166
setMsgLevel	166
setRealFmt	167
XPRBbasis	168
XPRBbasis	168
getCRef	168
isValid	168
reset	168
XPRBctr	170
XPRBctr	171
add	171
addTerm	172
delTerm	172
getAct	172
getCRef	173
getDual	173
getName	174
getRange	174
getRangeL	174
getRangeU	174
getRHS	175
getRNG	175
getRowNum	176
getSlack	176
getType	176
isModCut	177
isValid	177
print	177
reset	177
setModCut	177
setRange	178
setTerm	179
setType	179

XPRBcut	180
XPRBcut	181
add	181
addTerm	181
delTerm	182
getCRef	182
getID	182
getRHS	182
getType	183
isValid	183
print	183
reset	183
setID	184
setTerm	184
setType	184
XPRBexpr	186
XPRBexpr	187
add	187
assign	187
delTerm	188
getSol	188
mul	188
neg	189
setTerm	189
sqr	190
XPRBindexSet	191
XPRBindexSet	191
addElement	191
getCRef	192
getIndex	192
getIndexName	192
getName	193
getSize	193
isValid	193
print	194
reset	194
XPRBprob	195
XPRBprob	197
addCuts	197
clearDir	198
delCtr	198
delCut	198
delSos	199
exportProb	199
getCRef	199
getCtrByName	200
getIndexSetByName	200
getLPStat	200
getMIPStat	201
getName	201
getNumIIS	201
getObjVal	202
getProbStat	202
getSense	202
getSosByName	203
getVarByName	203
getXPRSprob	203
loadBasis	204

loadMat	204
maxim	205
minim	205
newCtr	206
newCut	206
newIndexSet	207
newSos	208
newVar	208
print	209
printObj	209
reset	210
saveBasis	210
setColOrder	210
setCutMode	211
setDictionarySize	211
setMsgLevel	212
setObj	212
setRealFmt	213
setSense	213
solve	213
sync	214
writeDir	215
XPRBrelation	216
XPRBrelation	216
getType	216
XPRBsos	217
XPRBsos	217
add	218
addElement	218
delElement	218
getCRef	219
getName	219
getType	219
isValid	219
print	220
setDir	220
XPRBvar	221
XPRBvar	222
fix	222
getColNum	222
getCRef	222
getLB	223
getLim	223
getName	223
getRCost	224
getRNG	224
getSol	225
getType	225
getUB	226
isValid	226
print	226
setDir	226
setLB	227
setLim	227
setType	228
setUB	228

6 BCL in Java	229
6.1 An overview of BCL in Java	229

6.1.1	Example	229
6.1.2	QCQP Example	233
6.1.3	Error handling	234
6.2	Java class reference	236
Appendix		238
A	BCL error messages	239
B	Using BCL with the Optimizer library	244
B.1	Switching between libraries	244
B.1.1	BCL-compatible Optimizer functions	244
B.1.2	Incompatible Optimizer functions	245
B.2	Initialization and termination	245
B.3	Loading the matrix	246
B.4	Indices of matrix elements	246
B.5	Using BCL-compatible functions	247
B.6	Using the Optimizer with BCL C++	248
B.7	Using the Optimizer with BCL Java	249
C	Working with cuts in BCL	251
C.1	Example	252
C.2	C++ version of the example	253
C.3	Java version of the example	253
Index		255

Chapter 1

Introduction

1.1 An overview of Xpress-BCL

The Xpress-BCL Builder Component Library provides an environment in which the Xpress-MP user may readily formulate and solve linear, mixed integer and quadratic programming models. Using BCL's extensive collection of functions, complicated models may be swiftly and simply constructed, preparing problems for optimization. Not merely limited to specific model construction, however, BCL's flexibility makes it the ideal engine for embedding in custom applications for the construction of generic modeling software. In combination with the Xpress-Optimizer, the two form a powerful combination.

Model formulation using Xpress-BCL is constraint-oriented. Such constraints may be built up either coefficient-wise, incrementally adding linear or quadratic terms until the constraint is complete, or through use of arrays of variables, constructing the constraint through a scalar product of variable and coefficient arrays. The former method allows for easier modification of models once constructed, whilst the latter enables swifter construction of new constraints.

BCL supports the full range of variable types available to users of the Xpress-Optimizer: continuous, semi-continuous, binary, semi-continuous integer, general and partial integer variables, as well as Special Ordered Sets of types 1 and 2 (SOS1 and SOS2). With additional functions for specifying directives to aid the global search, BCL enables preparation of every aspect of complicated (mixed) integer programming problems.

To complement the model construction routines, BCL supports a number of functions which allow a completed model to be passed directly to the Xpress-Optimizer, solved by the optimizer, and solution information reported back directly from BCL. For situations where the BCL solution functions do not provide enough capability to handle a particular user's requirements, problems may be manipulated using the Xpress-Optimizer library functions. Such close interactivity between BCL and the Xpress-Optimizer make these two libraries a perfect partnership.

BCL also supports a number of functions allowing easy input and output of model and solution data. In addition to a set of useful print functions, other functions also enable the export of constructed models as matrix files in a number of industry standard formats.

1.2 Note for Optimizer library users

BCL functions cover all aspects of modeling, and perform simple optimization tasks without making reference to the problem representation (matrix) used by the underlying solution algorithms. The more advanced Optimizer library user may nevertheless wish to access the problem matrix directly. It is possible to use all Optimizer library functions with the matrix generated by BCL. To this end, BCL provides several functions which specifically relate to the matrix representation.

The function `XPRBloadmat` explicitly transforms the constraint-wise representation in BCL into the matrix representation required by the Optimizer library. It is usually *not* necessary to call

this function because BCL automatically carries out this transformation whenever required.

The functions `XPRBgetcolnum` and `XPRBgetrownum` return the column and row indices associated with BCL variables and constraints respectively. While loading the matrix with a call to `XPRBloadmat`, all variables that do not occur in any constraint and all empty constraints are ignored and variable and constraint indices are updated correspondingly (with negative indices indicating that a variable or constraint is not part of the active matrix in the Optimizer).

It should be stressed that BCL, and thus the arrays storing references to problem variables, does *not* keep track of any changes to the matrix occurring during the solution procedure within the Optimizer. This implies that if linear presolve or integer preprocessing is used, the correct solution information is available only after the postsolve has been carried out. This is usually done automatically if the solution algorithm terminates correctly (see the description of `XPRBsolve` in Chapter 4 for details).

If the matrix is altered directly with Optimizer library functions such as `XPRSaddrows` or `XPRSchgcoef` it is *not* possible to retrieve the modifications in the BCL model. In order to maintain a coherent status, any such modification has to be carried out in BCL, followed by a call to function `XPRBloadmat`.

Appendix B explains in more detail how to use Optimizer library functions within a BCL program. Interested users are directed there for details

1.3 Structure of this manual

The main body of the manual is essentially organized into two parts. It begins in Chapter 2, with a brief overview of common BCL functions and their usage, covering model management, construction, solution and the output of information following optimization. These ideas are extended in Chapter 3, to cover some of the more advanced or less well-known features of the library. The use of index sets, special ordered sets and quadratic programming are all covered here.

Following the first two chapters, the remainder forms the main reference section of the manual. Chapter 4 details all functions in the library alphabetically, enabling swift access to information about function syntax and usage, accompanied by examples. This is followed in Chapter A by a list of BCL error and return codes. An overview of usage of BCL with the Xpress-Optimizer library and of the C++ and Java interfaces form the Appendices to the manual.

1.4 Conventions used

Throughout the manual standard typographic conventions have been used, representing computer code fragments with a *fixed width font*, whilst equations and equation variables appear in *italic type*. Where several possibilities exist for the library functions, those with C syntax have been used, and C style conventions have been used for structures such as arrays etc. Where appropriate, the following have also been employed:

- square brackets [...] contain optional material;
- curly brackets {...} contain optional material, of which one must be chosen;
- entities in *italics* which appear in expressions stand for meta-variables. The description following the meta-variable describes how it is to be used;
- the vertical bar symbol | is found on many keyboards as a vertical line with a small gap in the middle, but often confusingly displays on screen without the small gap in the middle. In UNIX it is referred to as the pipe symbol. Note that this symbol is not the same as the character sometimes used to draw boxes on a PC screen. In ASCII, the | symbol is 7C in hexadecimal, 124 decimal.

I. Modeling with BCL

Chapter 2

Modeling with BCL

2.1 Problem handling

2.1.1 Initialization and termination

Prototypes for all BCL functions are contained in the header file, `xprb.h`, which needs to be included at the top of any program which makes BCL function calls. The first stage in the model building process is to initialize BCL, either explicitly with a call to `XPRBinit` or implicitly by creating a new problem with function `XPRBnewprob` (see below). During its initialization BCL also initializes the Xpress-Optimizer, so if the two are to be used together, a separate call to `XPRSinit` is unnecessary. The initialization function checks for any necessary libraries, and runs security checks to determine license information about your Xpress-MP installation.

Once models have been constructed and BCL routines are no longer needed, the function `XPRBfree` may be called to reset BCL.

2.1.2 Problem creation and deletion

BCL has an object-oriented design. A mathematical model is represented in BCL by a problem that contains a collection of other objects (variables, constraints, index set etc). Every BCL function takes as the first argument the object it operates on.

A problem reference in BCL is a variable of type `XPRBprob`. A problem is created using the `XPRBnewprob` function, additionally providing a problem name, in the following way:

```
XPRBprob prob;  
...  
prob = XPRBnewprob("MyProb");
```

The problem reference, `prob`, is subsequently provided as the first argument to functions operating on the problem.

Once use of a particular problem has ended, the problem should be removed using `XPRBdelprob`, freeing associated resources. It should be noted that resources associated with problems are *not* released with a call to `XPRBfree`, so failure to explicitly delete each problem may result in memory leakage. It is also possible to delete just the solution information stored by BCL after an optimization run (including all problem-related information loaded in Xpress-Optimizer), if the definition of the problem is to be kept for later re-use but its solution data is not required any longer (function `XPRBresetprob`).

Note that for every BCL problem of type `XPRBprob` exists a corresponding Xpress-Optimizer problem (type `XPRSprob`). Although it is usually not necessary to access the optimizer problem directly in BCL programs, this may be required for certain advanced uses (see Appendix B for more detail).

Initialize a new model	<code>XPRBprob pb1;</code> <code>...</code> <code>pb1 = XPRBnewprob("Problem1");</code>
Delete problem definition	<code>XPRBdelprob(pb1);</code>
Delete solution information	<code>XPRBresetprob(pb1);</code>
Load problem matrix	<code>XPRBloadmat(pb1);</code>
Fix column ordering	<code>XPRBsetcolorder(pb1,1);</code>
Get problem name	<code>XPRBgetprobname(pb1);</code>

Figure 2.1: Creating, accessing and deleting problems in BCL

2.1.3 Other basic functions

Other functions are also useful for problem handling and manipulation. With `XPRBgetprobname`, the name for a particular problem specified by a reference may be obtained.

The function `XPRBloadmat` is really only needed by Optimizer library users. It explicitly transforms the BCL problem into the matrix representation in the Optimizer, passing the problem directly into the Optimizer. Usually this is done automatically by BCL whenever required, but it may be necessary to load the matrix without optimizing immediately, e.g. so that an advance basis can be loaded before starting the optimization. The matrix generated by BCL remains unchanged in repeated executions of the program; the column ordering criterion may be changed by setting the ordering flag to 1 (function `XPRBsetcolorder`) before the matrix is loaded.

2.1.4 Input and output settings

BCL supports a number of functions for directing the input and output of a program. Those functions are independent of the particular problem and consequently do not take the problem pointer as an argument or may be used with a `NULL` argument. They may be called prior to the creation of any problem using `XPRBnewprob`, and even prior to the initialization of BCL. Any other BCL function will result in an error if it is executed before BCL has been initialized.

Printout of BCL status information, warnings or error messages may be turned off (function `XPRBsetmsglevel`). With function `XPRBdefcbmsg`, the user may define the message callback function to intercept all output printed by BCL (including messages from the Optimizer library and output from the user's program printed with function `XPRBprintf`, the latter not being influenced by the setting of the message print level). Section 3.5 in the next chapter shows an example of a message callback.

The formatting of real numbers used by the BCL output functions (including matrix export) can be set with the function `XPRBsetrealfmt`.

For data input in BCL (using functions `XPRBreadline` and `XPRBreadarrline`), it is possible to switch from the (default) Anglo-American standard of using a decimal point to some other character, such as a decimal comma (`XPRBsetdecsign`).

2.1.5 Error handling

By default, BCL stops the program execution if an error occurs. With function `XPRBseterrctrl` the user may change this behavior: the error messages are still produced but the user's program has to provide the error handling. This setting may be useful, for instance, if an BCL program is embedded into some other application or executed under Windows.

Error handling by the user's program may either be implemented by checking the return values of all BCL functions, or preferably, by defining a callback (with function `XPRBdefcberr`) to intercept all warnings and errors produced by BCL. This function is not influenced by `XPRBsetmsglevel`, that is the user may turn off message printing and still be notified about any errors that occur. Section 3.5 in the next chapter shows an example of an error callback.

Set number format	<code>XPRBsetrealfmt(prob, "%8.4f");</code>
Set decimal sign	<code>XPRBsetdecsign(',', '');</code>
Set printout level	<code>XPRBsetmsglevel(prob, 1);</code>
Set error handling	<code>XPRBseterrctrl(0);</code>
Error handling callback	<code>void myerror(XPRBprob my_prob, void *my_object, int num, int type, const char *txt); XPRBdefcberr(prob, myerror, object);</code>
Printing callback	<code>void myprint(XPRBprob my_prob, void *my_object, const char *msgtext); XPRBdefcbmsg(prob, myprint, object);</code>
Get version number	<code>const char *version; version = XPRBgetversion();</code>

Figure 2.2: Input and output settings, and error handling in BCL

When reporting problems with the software, the user should always give the version of BCL. This information can be obtained with the function `XPRBgetversion`.

2.2 Variables

2.2.1 Basic functions

In BCL, variables are created one-by-one with a call to the function `XPRBnewvar`. These variables may belong to multi-dimensional arrays declared within C. Since one-dimensional arrays of variables are used as input to a number of functions, BCL also provides a specific object for this purpose, the type `XPRBarrvar`. This object stores a one-dimensional array of variables together with information about its size. That means such an array of variables may be used as a parameter to a function without having to specify its size separately. Details on specific functions for creating and accessing variable arrays are given in the following Section 2.2.2.

The length of variable names (like the names of all BCL objects) is unlimited. If no name is specified the system generates default names ("VAR" followed by an index). A name may occur repeatedly and, if so, BCL starts indexing the name, commencing with an index of 0.

All types of branching directives available in Xpress-MP can be set via the function `XPRBsetvardir`, including priorities, choice of the preferred branching direction and definition of pseudo costs. Bounds on variables are redefined by functions `XPRBsetub`, `XPRBsetlb`, `XPRBfixvar`, and `XPRBsetlim`. Function `XPRBsetlim` only applies to partial integer, semi-continuous and semi-integer variables, setting the lower bound of the continuous part or the semi-integer lower bound. Function `XPRBgetbyname` retrieves variables or arrays of variables via their name. Information on variables can be accessed with function `XPRBgetvarname`, `XPRBgetvartype`, `XPRBgetcolnum`, `XPRBgetbounds`, and `XPRBgetlim`. Function `XPRBsetvartype` changes the variable type. Figure 2.3 gives an overview of functions related to the creation, update and deletion of variables and arrays of variables.

2.2.2 Variable arrays

BCL provides a specific object for representing one-dimensional arrays of variables, as these are used as input to a number of functions. Variable arrays can be created either in one go, with a single function call to `XPRBnewarrvar`, or incrementally by copying single references to previously defined variables into an array of type `XPRBarrvar`.

If a variable array is created by a call to `XPRBnewarrvar`, all of the variables in the array receive the same type and bounds (these can be modified individually following creation). Otherwise, if the array is being defined incrementally, any previously defined variables (including elements of variable arrays) may be added to the array in an arbitrary order. In this case, the definition of the array is started by indicating its model name and size in `XPRBstartarrvar` and terminated by `XPRBendarrvar`. Entries can be positioned via `XPRBsetarrvarel` or simply placed

Creating variables	<pre> XPRBvar y, s[4]; y = XPRBnewvar(prob,XPRB_PL,"y",1,10); for (i=0;i<4;i++) s[i]=XPRBnewvar(prob,XPRB_UI,"st",1,10); </pre>
Creating variable arrays	<pre> XPRBarrvar av1, av2; av1=XPRBnewarrvar(prob,5,XPRB_SC,"a1",0,7); av2=XPRBstartarrvar(prob,3,"a2"); XPRBapparrvarel(av2,y); XPRBsetarrvarel(av2,2,s[3]); XPRBendarrvar(av2); </pre>
Accessing variables	<pre> double ubd, lbd, lim; XPRBgetvarname(y); XPRBgetvartype(s[1]); XPRBgetcolnum(av2[0]); XPRBgetbounds(y,&lbd,&ubd); XPRBgetlim(y,&lim); XPRBsetvartype(av1[1],XPRB_BV); </pre>
Accessing arrays	<pre> XPRBgetarrvarname(av2); XPRBgetarrvarsize(av1); </pre>
Delete a variable array	<pre> XPRBdelarrvar(av2); </pre>
Find by name	<pre> XPRBvar y1; XPRBarrvar a1; y1 = XPRBgetbyname(prob,"y",XPRB_VAR); a1 = XPRBgetbyname(prob,"a1",XPRB_ARR); </pre>
Branching directives	<pre> XPRBsetvardir(s[0],PR,1); XPRBclearidir(prob); </pre>
Setting bounds	<pre> XPRBsetlb(y,4); XPRBsetub(s[0],9); XPRBfixvar(av[2],6); XPRBsetlim(y,5); </pre>

Figure 2.3: Functions for creation, update, deletion and access of variables within BCL

at the first available free position by `XPRBapparrvarel`. For instance, assume we have defined four continuous variables $s[0], \dots, s[3]$ and a binary variable b . We may then wish to create an array av with the following three elements: $av[0] = b$, $av[1] = s[2]$, $av[2] = s[0]$. Regrouping different variables this way into a single data structure may help render the formulation of constraints or the access to information about model objects more transparent.

A variable may be copied into several arrays (function `XPRBsetarrvarel` or `XPRBapparrvarel`), but it is created only once as a variable or part of a variable array (using function `XPRBnewvar` or `XPRBnewarrvar`).

Function `XPRBgetbyname` retrieves arrays of variables via their name. It is also possible to obtain the name of an array (`XPRBgetarrvarname`) and its size, that is, the number of variables it contains (`XPRBgetarrvarsize`).

2.3 Constraints

2.3.1 Basic functions

Constraints are created either by a call to a specialized constraint function (see Section 2.3.2) or by subsequently adding all the desired terms to a constraint. In the latter case, a new constraint is started with function `XPRBnewctr` by indicating its type and (optionally) its name, variable and constant terms are added with functions `XPRBaddterm`, `XPRBsetterm` and `XPRBaddarrterm`. Function `XPRBaddterm` adds the indicated coefficient value to the coefficient of the variable, whereas `XPRBsetterm` overrides any previously defined coefficient for the variable in the constraint. It is also possible to add an entire array of variables at once to a constraint, together with the corresponding coefficients (function `XPRBaddarrterm`). Figure 2.4 gives some examples of constraint creation.

Since all functions for constraint definition identify the corresponding constraint via its model

$\sum_{i=0}^3 s_i \leq 20$ <pre>XPRBnewsum(prob, "S1", s, XPRB_L, 20);</pre>	<pre>XPRBctr ctr ctr = XPRBnewctr(prob, "S1", XPRB_L); for (i=0; i<=3; i++) XPRBaddterm(ctr, s[i], 1); XPRBaddterm(ctr, NULL, 20);</pre>
$\sum_{i=0}^3 D_i \cdot s_i = 9$ <pre>XPRBnewarrsum(prob, "S2", s, D, XPRB_E, 9);</pre>	<pre>ctr = XPRBnewctr(prob, "S2", XPRB_E); XPRBaddarrterm(ctr, s, D); XPRBaddterm(ctr, NULL, 9);</pre>
$s_0 + D_0 \leq y$ <pre>XPRBnewprec(prob, "Prc", s[0], D[0], y);</pre>	<pre>(s₀ - y ≤ -D₀) ctr=XPRBnewctr(prob, "Prc", XPRB_L); XPRBaddterm(ctr, s[0], 1); XPRBaddterm(ctr, y, -1); XPRBaddterm(ctr, NULL, -D[0]);</pre>

Figure 2.4: Constraint definition using the constraint functions provided by BCL (left column) or by adding coefficients (right column)

name, constraint definitions may be nested.

The length of constraint names is unlimited. If no name is specified the system generates default names ("CTR" followed by an index). A name may occur repeatedly and if so, BCL starts indexing the name, commencing with an index of 0. Variables and variable arrays used in the definition of a constraint must be defined previously. Any other variables not occurring in this constraint may be defined later in the model.

After a constraint has been defined, its type may be changed to a range constraint by indicating the lower and upper bounds in a call to function `XPRBsetrange`. Function `XPRBgetbyname` retrieves constraints via their name.

A coefficient can be deleted with `XPRBdelterm`, or an entire constraint definition by `XPRBdelctr`. It is possible to retrieve the constraint name (`XPRBgetctrname`), the matrix row index (`XPRBgetrownum`), the constraint type (`XPRBgetctrtype`), the range values (`XPRBgetrange`, only applicable to ranged constraints) and right hand side value (`XPRBgetrhs`), as well as changing the constraint type (`XPRBsetctrtype`). A constraint can be transformed into a model cut (`XPRBsetmodcut`) and function `XPRBgetmodcut` indicates whether a constraint has been defined as a model cut.

In addition to the functions for handling linear constraints listed here, BCL also lets you define quadratic constraints for the formulation of QP and QCQP problems, see Section 3.4 for further detail.

2.3.2 Predefined constraint functions

Besides the functions described above for defining constraints incrementally, BCL also provides some predefined constraint functions for formulating constraints 'in one go'. The function `XPRBnewarrsum` creates a standard linear constraint with the indicated coefficients. The function `XPRBnewsum` creates a straight sum of the variables with each coefficient set to one. The function `XPRBnewprec` creates a so-called *precedence constraint* in which a variable plus a constant are less than or equal to a second variable (typically, this relation is established between start time variables in scheduling problems, hence the name).

2.3.3 Objective function

The objective function (Figure 2.5) may be interpreted as a special type of constraint. It is defined like any other constraint, usually choosing the constraint type `XPRB_N`. But it is also possible to take a constraint of any other type. In the latter case, the variable terms of the constraint form the objective function but the equation or inequality expressed by the constraint also remains part of the problem. The objective function is declared via functions `XPRBsetobj`. If a different objective has been defined previously, it is replaced by the new choice.

Set objective function	<code>XPRBctr c; XPRBsetobj(prob,c);</code>
Set objective sense	<code>XPRBsetsense(prob,XPRB_MAXIM);</code>
Access objective sense	<code>int dir; dir = XPRBgetsense(prob);</code>
Locate constraint	<code>XPRBctr c; c = XPRBgetbyname(prob,"Sum1",XPRB_CTR);</code>
Define range constraint	<code>XPRBsetrange(c,1,5,15);</code>
Delete a constraint	<code>XPRBdelctr(c);</code>
Delete a constraint term	<code>XPRBvar y; XPRBdelterm(c,y);</code>
Accessing constraints	<code>double bdl, bdu; XPRBgetctrname(c); XPRBgetrange(c,&bdl,&bdu); XPRBgetrownum(c); XPRBgetctrtype(c); XPRBsetctrtype(c,XPRB_L); XPRBgetmcut(c); XPRBsetmcut(c,1);</code>

Figure 2.5: Defining the objective function and functions for modifying and accessing constraints

Solve active problem	<code>XPRBsolve(prob,"dg"); XPRBminim(prob,"pl"); XPRBmaxim(prob,"");</code>
Status information	<code>XPRBgetprobstat(prob); XPRBgetlpstat(prob); XPRBgetmipstat(prob);</code>
Get objective value	<code>XPRBgetobjval(prob);</code>
Solution information	<code>XPRBvar y; XPRBctr c; XPRBgetsol(y); XPRBgetdual(c); XPRBgetrcost(y); XPRBgetslack(c); XPRBgetact(c);</code>
Ranging information	<code>XPRBgetvarrng(y,XPRB_UCOST); XPRBgetctrrng(c,XPRB_LOACT);</code>
Advanced bases	<code>XPRBbasis b; b=XPRBsavebasis(prob); XPRBloadbasis(b); XPRBdelbasis(b);</code>

Figure 2.6: Solving and solution information

The sense of the objective function can be set to be minimization (default) or maximization with function `XPRBsetsense`. Function `XPRBgetsense` returns the sense of the objective function.

All solution functions (`XPRBsolve`, `XPRBminim`, `XPRBmaxim`) and the problem output with `XPRBexportprob` require the objective to be defined. If the sense of the optimization has not been set, the problem is minimized by default.

2.4 Solving and solution information

As well as enabling model definition, BCL also provides common solving and solution information functions, as summarized in Figure 2.6. For more advanced tasks the user may employ the corresponding Optimizer library functions, once the matrix has been loaded into the Optimizer (function `XPRBloadmat`). However, only the BCL functions can reference the BCL model objects when retrieving the solution information.

Before any solution function is called, the objective function must be selected using `XPRBsetobj`. The function `XPRBsolve` also requires the sense of the objective to be set, that is, whether to minimize (default) or to maximize the objective. All solution functions `XPRBsolve`,

`XPRBminim`, and `XPRBmaxim` can be parameterized to choose the type of solution algorithm. Once the problem has been solved, the following solution information can be obtained: the optimal objective function value (`XPRBgetobjval`), values for all the problem variables (`XPRBgetsol`), slack values (`XPRBgetslack`), reduced costs (`XPRBgetrcost`), constraint activity (`XPRBgetact`), and dual values (`XPRBgetdual`). It is also possible to obtain ranging information for variables (`XPRBgetvarrng`) and constraints (`XPRBgetctrng`) after solving an LP problem.

If the objective function value or solution information for variables or constraints is accessed during the optimization (for instance from Xpress-Optimizer callbacks) the solution information in BCL needs to be updated with a call to `XPRBsync` with the parameter `XPRB_XPRS_SOL` (see Appendix B for more detail).

Before solving or accessing solution information it may be helpful to check the current problem and/or solution status (using functions `XPRBgetprobstat`, `XPRBgetlpstat` and `XPRBgetmipstat`). It may happen that a variable defined in the model does not appear in any constraint, or a constraint only contains 0-valued coefficients so that is ignored when loading the problem into the Optimizer. In these cases the object's column or row index is negative and no solution information can be obtained.

With BCL, it is also possible to save the current basis of a problem in memory and reload (and/or delete) it after some changes have been carried out to the problem. These changes may include, for instance, the addition or deletion of variables and constraints.

For more advanced functionality using Optimizer library functions refer to the Optimizer Reference Manual.

2.5 Example

The following example is an extract of a scheduling problem: four jobs with different durations need to be scheduled with the objective to minimize the makespan (= completion time of the last job). The complete model also includes resource constraints that are omitted here for clarity's sake. For every job j its duration DUR_j is given. We define decision variables $start_j$ representing the start time of jobs and binary variables $delta_{jt}$ indicating whether job j starts in time period t ($delta_{jt} = 1$). We also define a variable z for the maximum makespan. The makespan can be expressed as a 'dummy job' of duration 0 that is the successor of all other jobs (constraints *Makespan* in the model below). We also formulate a precedence relation between two jobs (constraint *Prec*). The start time variables need to be linked to the binary variables (constraints *Link*). And finally, the binary variables are used to express that every job has a unique start time (constraints *One*).

2.5.1 Model formulation using basic functions

```
#include <stdio.h>
#include "xprb.h"

#define NJ      4          /* Number of jobs */
#define NT     10         /* Time limit */

double DUR[] = {3,4,2,2}; /* Durations of jobs */
XPRBvar start[NJ];        /* Start times of jobs */
XPRBvar delta[NJ][NT];   /* Binaries for start times */
XPRBvar z;               /* Max. completion time */
XPRBprob prob;           /* BCL problem */

void jobs_model(void)
{
    XPRBctr ctr;
    int j,t;

    prob=XPRBnewprob("Jobs"); /* Initialization */

    for(j=0;j<NJ;j++)          /* Create start time variables */
        start[j] = XPRBnewvar(prob, XPRB_PL, "start", 0, NT);
```

```

z = XPRBnewvar(prob, XPRB_PL, "z", 0, NT); /* Makespan var. */

for(j=0;j<NJ;j++) /* Declare binaries for each job */
    for(t=0;t<(NT-DUR[j]+1);t++)
        delta[j][t] = XPRBnewvar(prob, XPRB_BV, "delta", 0, 1);

for(j=0;j<NJ;j++) /* Calculate max. completion time */
    XPRBnewprec(prob, "Makespan", start[j], DUR[j], z);
/* Precedence relation betw. jobs */
XPRBnewprec(prob, "Prec", start[0], DUR[0], start[2]);

for(j=0;j<NJ;j++) /* Linking start times & binaries */
{
    ctr = XPRBnewctr(prob, "Link", XPRB_E);
    for(t=0;t<(NT-DUR[j]+1);t++)
        XPRBaddterm(ctr, delta[j][t], t+1);
    XPRBaddterm(ctr, start[j], -1);
}

for(j=0;j<NJ;j++) /* Unique start time for each job */
{
    ctr = XPRBnewctr(prob, "One", XPRB_E);
    for(t=0;t<(NT-DUR[j]+1);t++) XPRBaddterm(ctr, delta[j][t], 1);
    XPRBaddterm(ctr, NULL, 1);
}

ctr = XPRBnewctr(prob, "OBJ", XPRB_N);
XPRBaddterm(ctr, z, 1);
XPRBsetobj(prob, ctr); /* Set objective function */

/* Upper bounds on start time variables */
for(j=0;j<NJ;j++) XPRBsetub(start[j], NT-DUR[j]+1);
}

```

2.5.2 Using variable arrays

In the subsequent code, we replace the variables $start_j$ and $delta_{jt}$ by arrays of variables $start$ and $delta_j$. Note that the variables can still be addressed in the same way as before. The main advantage of this formulation is that now some of the predefined constraint functions may be used in the model definition. Changes to the previous version are highlighted in bold.

```

#include <stdio.h>
#include "xprb.h"

#define NJ      4          /* Number of jobs */
#define NT     10         /* Time limit */

double DUR[] = {3,4,2,2}; /* Durations of jobs */
XPRBarrvar start;          /* Start times of jobs */
XPRBarrvar delta[NJ];     /* Sets of binaries */
XPRBvar z;                 /* Maxi. completion time */
XPRBprob prob;             /* BCL problem */

void jobs_model_array(void)
{
    XPRBctr ctr;
    int j,t;
    double c[NT];

    prob=XPRBnewprob("Jobs"); /* Initialization */

    /* Create start time variables */
    start = XPRBnewarrvar(prob, NJ, XPRB_PL, "start", 0, NT);
    z = XPRBnewvar(prob, XPRB_PL, "z", 0, NT); /* Makespan var. */

    for(j=0;j<NJ;j++) /* Set of binaries for each job */
        delta[j] = XPRBnewarrvar(prob, (NT-(int)(DUR[j])+1), XPRB_BV,
            "delta", 0, 1);

    for(j=0;j<NJ;j++) /* Calculate max. completion time */
        XPRBnewprec(prob, "Makespan", start[j], DUR[j], z);
}

```

```

/* Precedence relation betw. jobs */
XPRBnewprec(prob, "Prec", start[0], DUR[0], start[2]);

for(j=0; j<NJ; j++) /* Linking start times & binaries */
{
    ctr = XPRBnewctr(prob, "Link", XPRB_E);
    for(t=0; t<(NT-DUR[j]+1); t++) c[t]=t+1;
    XPRBaddarrterm(ctr, delta[j], c);
    XPRBaddterm(ctr, start[j], -1);
}
/* Alternative constraint formulation:
for(j=0; j<NJ; j++)
{
    ctr = XPRBnewsumc(prob, "Link", delta[j], 1, XPRB_E, 0);
    XPRBaddterm(ctr, start[j], -1);
}
*/

for(j=0; j<NJ; j++) /* Unique start time for each job */
{
    ctr = XPRBnewctr(prob, "One", XPRB_E);
    for(t=0; t<(NT-DUR[j]+1); t++) XPRBaddterm(ctr, delta[j][t], 1);
    XPRBaddterm(ctr, NULL, 1);
}

ctr = XPRBnewctr(prob, "OBJ", XPRB_N);
XPRBaddterm(ctr, z, 1);
XPRBsetobj(prob, ctr); /* Set objective function */

/* Upper bounds on start time variables */
for(j=0; j<NJ; j++) XPRBsetub(start[j], NT-DUR[j]+1);
}

```

The set of constraints *Link* (linking start time variables and binaries) can also be formulated using arrays and the constraint relation `XPRBnewarrsum`. These arrays are created by copying references to previously defined variables. In the example below, they serve only to create this set of constraints so that there is no need for storing them. If these arrays were to be used later on, they should be given different names, perhaps using an array `av[NJ]`.

Note that the example below works with both formulations of the model, using single variables or arrays of variables for start times `start` and indicator variables `delta`.

```

for(j=0; j<NJ; j++)
{
    double ind[NT];
    v = XPRBstartarrvar(prob, NT-(int)(DUR[j])+2, "v1");
    /* Define an array of size NT-DUR[j]+2 */
    for(t=0; t<(NT-(int)(DUR[j])+1); t++)
    {
        XPRBapparrvarel(v, delta[j][t]); /* Add a variable to v */
        ind[t]=t+1; /* Add a coefficient */
    }
    XPRBapparrvarel(v, start[j]); /* Add "start" variable */
    XPRBendarrvar(v); /* Terminate array def. */
    ind[NT-(int)(DUR[j])+1]=-1; /* Add another coeff. */
    XPRBnewarrsum(prob, "Link", v, ind, XPRB_E, 0);
    /* Def. constraint using array v */
    XPRBdelarrvar(v); /* Free the allocated memory */
}

```

2.5.3 Completing the example: problem solving and output

We now want to solve the example problem and retrieve the solution values (objective function and start times of all jobs). We do this with a separate function, `jobs_solve`. To complete the program we write a main that calls the model definition and the solution functions.

```

void jobs_solve(void)
{
    int statmip;
    int j;

```

```

XPRBsetsense(prob, XPRB_MINIM);
XPRBsolve(prob, "g");          /* Solve the problem as MIP */
statmip = XPRBgetmipstat();    /* Get the problem status */
if((statmip == XPRB_MIP_SOLUTION) ||
    (statmip == XPRB_MIP_OPTIMAL))
{
    /* An integer solution has been found */
    printf("Objective: %g\n", XPRBgetobjval());
    for(j=0; j<NJ; j++)
        printf("%s: %g\n", XPRBgetvarname(s[j]), XPRBgetsol(s[j]));
    /* Print out the solution for all start times */
}
}

int main(int argc, char **argv)
{
    jobs_model();                /* Problem definition */
    jobs_solve();                /* Solve and print solution */
    return 0;
}

```

If we want to influence the branch-and-bound tree search, we may try setting some branching directives. To prioritize branching on variables that represent early start times the following lines can be added to csolve before the solution algorithm is started.

```

for(j=0; j<NJ; j++)
    for(t=0; t<NT-DUR[j]+1; t++)
        XPRBsetvardir(delta[j][t], XPRB_PR, 10*(t+1));
    /* Give highest priority to var.s for earlier start times */

```

Chapter 3

Further modeling topics

3.1 Data input and index sets

BCL requires the user to read data into their own structures or data arrays by using standard C functions for accessing data files. The functions `XPRBreadarrline` and `XPRBreadline` read data from data files in the `diskdata` format (see the documentation of the module `mmetc` in the *Xpress-Mosel Language Reference Manual* for details). The first function reads (dense) data tables with all entries of the same type, the second reads tables with items of different types (such as text strings and numbers). In particular, `XPRBreadline` is well suited to read sparse data tables that are indexed by so-called *index sets*. Roughly speaking, an index set is a set of items such as text strings that index data tables and other objects in the model in a clearer way than numerical values (for details refer to the *Xpress-Mosel Reference Manual*).

A new index set is created by calling function `XPRBnewidxset`. Set elements are added with function `XPRBaddidxel`. An element of a set can be retrieved either by its name (`XPRBgetidxel`) or by its order number within the set (using the function `XPRBgetidxelname`). A data item may be part of several index sets. Function `XPRBgetidxsetsize` returns the current size (*i.e.* the number of set elements) of an index set.

The definition of index sets may be nested, that is while reading a data file the user may fill up several index sets at a time. The size of index sets grows automatically as required. The user sets some initial size at the creation of the set, but if less elements are added the size returned by `XPRBgetidxsetsize` will be smaller than this value and if more elements are added the size is increased accordingly.

3.1.1 Example

Taking the program example from the previous chapter, we now assume that we want to give names to the jobs, such as ABC14, DE45F, GH9IJ99, KLMN789. We further assume that these

Data input from file	<pre>FILE *datafile; char name[50]; double dval, dvals[5]; XPRBreadlinecb(XPRB_FGETS, datafile, 200, "T,d", name, &dval); XPRBreadarrlinecb(XPRB_FGETS, datafile, 200, "d;", dvals, 5);</pre>
Create a new index set	<pre>XPRBidx set1; set1 = XPRBnewidxset(prob, "Set1", 100);</pre>
Add index to a set	<pre>XPRBaddidxel(set1, "Prob1");</pre>
Accessing index sets	<pre>int size, ind; ind = XPRBgetidxel(set1, "Prod1"); name = XPRBgetidxelname(set1, 14); name = XPRBgetidxsetname(set1); size = XPRBgetidxsetsize(set1);</pre>

Figure 3.1: Data input from file and accessing index sets: creation of sets, addition of elements, retrieving elements, and the index set size.

names, together with the durations, are given in a separate data file, durations.dat:

```
ABC14, 3
DE45F, 4
GH9IJ99, 2
KLMN789, 2
```

If data is read with function `XPRBreadline`, it is possible to use comments (preceded by `!`) and line continuation signs (`&`) in the data file. (Note that single strings and numbers may not be written over several lines.) The input function also skips blanks and empty lines. If separator signs other than blanks are used, the value 0 may be omitted in the data file (for instance, a data line `0, 0, 0` could as well be written as `, ,` or, using blanks as separators, `0 0 0`). The following is functionally equivalent to the contents of durations.dat:

```
ABC14, 3      ! product1, duration1
DE45F, &      ! this line is continued
4             ! in the next line
  GH9IJ99, 2   ! blanks are skipped
              ! as well as empty lines
KLMN789, 2
```

Separating the input data from the definition allows the same model to be rerun with different data sets without having to recompile the program code. To accommodate data in this form the model program must be written or edited appropriately. In the following program, a function for data input is added to the code seen in the previous chapter. The space for the decision variable arrays is allocated once the array sizes are known. Notice that we use the job names as the names of the decision variables.

```
#include <stdio.h>
#include <stdlib.h>
#include "xprb.h"
#define MAXNJ 4          /* Maximum number of jobs */
#define NT 10            /* Time limit */
int NJ=0;                /* Number of jobs read in */
double DUR[MAXNJ];       /* Durations of jobs */

XPRBidxset Jobs;          /* Job names */
XPRBvar *start;           /* Start times of jobs */
XPRBvar **delta;          /* Binaries for start times */
XPRBvar z;                /* Max. completion time */
XPRBprob prob;            /* BCL problem */

void read_data(void)
{
    char name[100];
    FILE *datafile;
    Jobs = XPRBnewidxset(prob, "jobs", MAXNJ);
                                /* Create a new index set */
    datafile=fopen("durations.dat", "r");
                                /* Open data file for read access */
    while(NJ<MAXNJ) &&
        XPRBreadlinecb(XPRB_FGETS, datafile, 99, "T,d", name, &DUR[NJ])
    { /* Read in all (non-empty) lines up to the end of the file */
        XPRBaddidxel(Jobs, name); /* Add job to the index set */
        NJ++;
    }
    fclose(datafile);           /* Close the input file */
    printf("Number of jobs read: %d\n", XPRBgetidxsetsize(Jobs));
}

void jobs_model(void)
{
    XPRBctr ctr;
    int j,t;
                                /* Create start time variables with bounds */
    start = (XPRBvar *)malloc(NJ * sizeof(XPRBvar));
    if(start==NULL)
    { printf("Not enough memory for 'start' variables.\n");
      exit(0); }
}
```



```

for(j=0; j<NJ; j++)
    start[j] = XPRBnewvar(prob, XPRB_PL, "start", 0, NT-DUR[j]+1);
z = XPRBnewvar(prob, XPRB_PL, "z", 0, NT); /* Makespan var. */
/* Declare binaries for each job */
delta = (XPRBvar **)malloc(NJ * sizeof(XPRBvar*));
if(delta==NULL)
{ printf("Not enough memory for 'delta' variables.\n");
  exit(0); }
for(j=0; j<NJ; j++)
{
    delta[j] = (XPRBvar *)malloc(NT* sizeof(XPRBvar));
    if(delta[j]==NULL)
    { printf("Not enough memory for 'delta_j' variables.\n");
      exit(0); }
    delta[j][t] = XPRBnewvar(XPRB_BV,
        XPRBnewname("delta%s_%d", XPRBgetidxelname(Jobs, j), t+1),
        0, 1);
}

for(j=0; j<NJ; j++) /* Calculate max. completion time */
    XPRBnewprec(prob, "Makespan", start[j], DUR[j], z);
/* Precedence relation betw. jobs */
XPRBnewprec(prob, "Prec", start[0], DUR[0], start[2]);

for(j=0; j<NJ; j++) /* Linking start times & binaries */
{
    ctr = XPRBnewctr(prob, "Link", XPRB_E);
    for(t=0; t<(NT-DUR[j]+1); t++)
        XPRBaddterm(ctr, delta[j][t], t+1);
    XPRBaddterm(ctr, start[j], -1);
}

for(j=0; j<NJ; j++) /* Unique start time for each job */
{
    ctr = XPRBnewctr(prob, "One", XPRB_E);
    for(t=0; t<(NT-DUR[j]+1); t++) XPRBaddterm(ctr, delta[j][t], 1);
    XPRBaddterm(ctr, NULL, 1);
}

ctr = XPRBnewctr(prob, "OBJ", XPRB_N);
XPRBaddterm(ctr, z, 1);
XPRBsetobj(prob, ctr); /* Set objective function */

jobs_solve(); /* Solve the problem */

free(start);
for(j=0; j<NJ; j++) free(delta[j]);
free(delta);
}

int main(int argc, char **argv)
{
    prob=XPRBnewprob("Jobs"); /* Initialization */
    read_data(); /* Read data from file */
    jobs_model(); /* Define & solve the problem */
    return 0;
}

```

3.2 Special Ordered Sets

3.2.1 Basic functions

Special Ordered Sets of type n ($n=1, 2$) are sets of variables of which at most n may be non-zero at an integer feasible solution. Associated with each set member is a real number (weight), establishing an ordering among the members of the set. In SOS of type 2, any positive variables must be adjacent in the sense of this ordering.

In BCL, Special Ordered Sets may be defined in different ways as illustrated in Figure 3.2. As with arrays and constraints, they may be created either with a call to a single function (see

	<pre> XPRBsos set1, set2; XPRBarvar s; </pre>
Immediate (ref. constraint)	<pre> XPRBctr c; set1=XPRBnewsosrc(prob, "sA", XPRB_S2, s, c); </pre>
Immediate (coefficients)	<pre> double C[] = 1,2,3,4; set2=XPRBnewsosw(prob, "sB", XPRB_S1, s, C); </pre>
Consecutive definition	<pre> set2=XPRBnewsos(prob, "sB", XPRB_S1); XPRBaddsosarrel(set2, s, C); </pre>
Delete set definition	<pre> XPRBdelsos(set2); </pre>
Accessing sets	<pre> XPRBaddsosel(set2, s[2], 4, 5); XPRBdelsosel(set1, s[0]); XPRBgetsosname(set1); XPRBgetsostype(set2); </pre>

Figure 3.2: Defining and accessing SOS: immediate (single function) by indicating a reference constraint; or consecutive definition by adding coefficients for all members.

Section 3.2.2), or by adding coefficients consecutively.

In the basic, incremental definition, function `XPRBnewsos` marks the beginning of the definition of a set. Single members are added by function `XPRBaddsosel` and arrays by function `XPRBaddsosarrel`, each time indicating the corresponding coefficients. Single elements, or an entire set definition, can be deleted with functions `XPRBdelsosel` and `XPRBdelsos` respectively. BCL also has functions to retrieve the name of a SOS and its type (`XPRBgetsosname` and `XPRBgetsostype`). It is also possible to set branching directives for a SOS (function `XPRBsetsosdir`), including priorities, choice of the preferred branching direction and definition of pseudo costs.

3.2.2 Array-based SOS definition

BCL provides two functions for creating Special Ordered Sets with a single function call: `XPRBnewsosrc` and `XPRBnewsosw`. With both functions, a new SOS is created by indicating the type (1 or 2), an array of variables and the corresponding weight coefficients for establishing an ordering among the set elements. With `XPRBnewsosrc`, these coefficients are taken from the variables' coefficients in the indicated reference constraint. When using function `XPRBnewsosw`, the user directly provides an array of weight coefficients.

3.2.3 Example

In the previous examples, instead of defining the delta variables as binaries, the problem can also be formulated using SOS of type 1. In this case, the delta variables are defined to be continuous as the SOS1 property and their unit sum ensure that one and only one takes the value one.

```

XPRBprob prob; /* BCL problem */
XPRBvar delta[NJ][NT]; /* Variables for start times */
XPRBsos set[NJ];

void jobs_model(void)
{
    ...
    for(j=0; j<NJ; j++) /* Declare a variable for each job */
        for(t=0; t<NT-DUR[j]+1; t++) /* and for each start time */
            delta[j][t] = XPRBnewvar(prob, XPRB_PL,
                                    XPRBnewname("delta%d%d", j+1, t+1), 0, 1);

    for(j=0; j<NJ; j++)
    {
        /* Create a new SOS1 */
        set[j] = XPRBnewsos(prob, "sosj", XPRB_S1);
        for(t=0; t<NT-D[j]+1; t++) /* Add variables to the SOS */
            XPRBaddsosel(set[j], delta[j][t], t+1);
    }
}

```

In order to simplify the definition of the SOS one can use the model formulation with variable arrays presented in the previous chapter. The constraints *Link* are employed as the reference constraints to determine the weight coefficient for each variable (the constraints need to be stored in an array, *Link*).

```
XPRBprob prob; /* BCL problem */
XPRBarrvar delta[NJ]; /* Sets of var.s for start times */
XPRBsos set[NJ];

void jobs_model(void)
{
    XPRBctr Link[NJ]; /* "Link" constraints */
    ...
    for(j=0;j<NJ;j++) /* Declare a set of var.s for each job */
        delta[j] = XPRBnewarrvar(prob, (NT-(int)DUR[j]+1), XPRB_PL,
                                XPRBnewname("delta%d",j+1), 0, 1);

    for(j=0;j<NJ;j++) /* Linking start times & binaries */
    {
        Link[j] = XPRBnewsumc(prob,"Link",delta[j],1,XPRB_E,0);
        XPRBaddterm(Link[j],start[j],-1);
    }

    /* Create a SOS1 for each job using constraints "Link" as
       reference constraints */
    for(j=0;j<NJ;j++)
        set[j] = XPRBnewsosrc(prob, "sosj", XPRB_S1, delta[j], Link[j]);
}
```

Instead of setting directives on the binary variables, we may now define branching directives for the SOS1.

```
for(j=0;j<NJ;j++) XPRBsetsosdir(set[j],XPRB_DN,0);
/* First branch downwards on sets */
```

3.3 Output and printing

BCL provides printing functions for variables, constraints, Special Ordered Sets, and index sets (`XPRBprintvar`, `XPRBprintarrvar`, `XPRBprintctr`, `XPRBprintsos`, `XPRBprintidxset`) as well as the entire model definition (`XPRBprintprob`). Any program output may be printed with `XPRBprintf` in a similar way to the C function `printf`. The output of all functions mentioned above is intercepted by the callback `XPRBdefcbmsg` if this function has previously been defined by the user.

It is also possible to output the problem to a file in extended LP format or as a matrix in extended MPS format (function `XPRBexportprob`). Note that unlike standard LP format, the extended LP format supports Special Ordered Sets and non-standard variable types (semi-continuous, semi-integer, or partial integers). Like the standard LP format it requires the sense of the objective function to be defined.

3.3.1 Example

We may now augment the last few lines of the model definition (`cmodel` or `cmodel_array`) of our example with some output functions. Note that these output functions may be added at any time to print the current problem definition in BCL. The function `XPRBprintprob` prints the complete BCL problem definition to the standard output. The function `XPRBexportprob` writes the problem definition in LP format or as a matrix in extended MPS format to the indicated file.

```
XPRBprintprob(prob); /* Print out the problem definition */
XPRBexportprob(prob,XPRB_MPS,"expl1");
/* Output matrix to MPS file */
```

File output	<code>XPRBexportprob(prob,XPRB_MPS,"expl2");</code>
Print model objects	<code>XPRBvar y;</code> <code>XPRBprintvar(y);</code> <code>XPRBarrvar av;</code> <code>XPRBprintarrvar(av);</code> <code>XPRBctr c;</code> <code>XPRBprintctr(c);</code> <code>XPRBsos s;</code> <code>XPRBprintsos(s);</code> <code>XPRBidxset is;</code> <code>XPRBprintidxset(is);</code>
Print a given problem	<code>XPRBprintprob(prob);</code>
Print program output	<code>XPRBprintf("Print this text");</code>
Compose a name string	<code>int i = 3;</code> <code>XPRBnewname("abc%d",i);</code>

Figure 3.3: File output and printing.

Instead of printing the entire problem with function `XPRBprintprob`, it is also possible to display single variables or constraints as soon as they have been defined. The following modified extract of the model definition may serve as an example.

```
#include <stdio.h>
#include "xprb.h"

#define NJ      4          /* Number of jobs */
#define NT     10         /* Time limit */

double DUR[] = {3,4,2,2}; /* Durations of jobs */
XPRBvar start[NJ];        /* Start times of jobs */
XPRBprob prob;            /* BCL problem */
...
void cmodel(void)
{
    XPRBctr ctr;
    int j,t;

    prob=XPRBnewprob("Jobs"); /* Initialization */

    for(j=0;j<NJ;j++)         /* Create start time variables */
    {
        start[j] = XPRBnewvar(prob,XPRB_PL,"start",0,NT);
        XPRBprintvar(start[j]);
        XPRBprintf(" ", " ");
    }
    ...
    /* Precedence relation betw. jobs */
    ctr = XPRBnewprec(prob,"Prec",start[0],DUR[0],start[2]);
    XPRBprintctr(ctr);
    ...
}
```

3.4 Quadratic Programming with BCL

As an extension to LP and MIP, BCL also provides support for formulating and solving Quadratic Programming (QP) and Mixed Integer Quadratic Programming (MIQP) problems, that is, problems with linear constraints with a quadratic objective function of the form

$$c^T x + x^T Q x$$

where x is the vector of decision variables, c is the cost vector, and Q is the quadratic cost coefficient matrix. The matrix Q must be symmetric. It should also be positive semi-definite if the problem is to be minimized, and negative semi-definite if it is to be maximized, because

Add quadratic term	<code>XPRBctr c; XPRBvar x1; XPRBaddqterm(c, x1, x1, 3);</code>
Set quadratic term	<code>XPRBvar x2; XPRBsetqterm(c, x1, x2, -7.2);</code>
Delete a quadratic term	<code>XPRBdelqterm(c, x2, x1);</code>

Figure 3.4: Defining and accessing quadratic terms in BCL.

the Xpress-Optimizer solves convex QP problems. If the problem is not convex, the solution algorithms may not converge at all, or may only converge to a locally optimal solution.

Release 4.0 of BCL extends this functionality to Quadratically Constrained Quadratic Programming (QCQP) problems, that is, problems that in addition to a quadratic objective function have constraints of the form

$$a^T x + x^T Q x \leq b$$

where a is the coefficient vector for the linear terms, b the constant RHS value, and the same conditions as in objective functions apply to the quadratic coefficient matrix Q (positive semi-definite in \leq constraints, and negative semi-definite in \geq constraints). Quadratic constraints in QCQP problems must be inequalities.

In BCL, the quadratic part of constraints is defined termwise, much like what we have seen for the definition of linear constraints in Section 2.3. The coefficient of a quadratic term is either set to a given value (`XPRBsetqterm`) or its value is augmented by the given value (`XPRBaddqterm`). Quadratic objective functions are set in the same way as linear ones with a call to `XPRBsetobj`. Note that the definition of the quadratic constraint terms should always be preceded by the definition of the corresponding variables.

Unless BCL is used in Student Mode, functions `XPRBprintprob`, `XPRBprintobj`, `XPRBexportprob`, and `XPRBprintctr` will print or output to a file the complete problem / constraint definition, including the quadratic terms.

3.4.1 Example

We wish to distribute a set of points represented by tuples of x-/y-coordinates on a plane minimizing the total squared distance between all pairs of points. For each point i we are given a target location (CX_i, CY_i) and the (square of the) maximum allowable distance R_i to this location.

In mathematical terms, we have two decision variables x_i and y_i for the coordinates of every point i . The objective to minimize the total squared distance between all points is expressed by the following sum.

$$\sum_{i=1}^{N-1} \sum_{j=i+1}^N (x_i - x_j)^2 + (y_i - y_j)^2$$

For every point i we have the following quadratic inequality.

$$(x_i - CX_i)^2 + (y_i - CY_i)^2 \leq R_i$$

The following BCL program (`xbairport.c`) implements and solves this problem.

```
#include <stdio.h>
#include "xprb.h"

#define N 42
double CX[N], CY[N], R[N];

...                               /* Initialize the data arrays */
```

```

int main(int argc, char **argv)
{
    int i, j;
    XPRBprob prob;
    XPRBvar x[N], y[N];          /* x-/y-coordinates to determine */
    XPRBctr cobj, c;

    prob=XPRBnewprob("airport"); /* Initialize a new problem in BCL */

    /**** VARIABLES ****/
    for(i=0;i<N;i++)
        x[i] = XPRBnewvar(prob, XPRB_PL, XPRBnewname("x(%d)",i), -10, 10);
    for(i=0;i<N;i++)
        y[i] = XPRBnewvar(prob, XPRB_PL, XPRBnewname("y(%d)",i), -10, 10);

    /****OBJECTIVE****/
    /* Minimize the total distance between all points */
    cobj = XPRBnewctr(prob, "TotDist", XPRB_N);
    for(i=0;i<N-1;i++)
        for(j=i+1;j<N;j++)
        {
            XPRBaddqterm(cobj, x[i], x[i], 1);
            XPRBaddqterm(cobj, x[i], x[j], -2);
            XPRBaddqterm(cobj, x[j], x[j], 1);
            XPRBaddqterm(cobj, y[i], y[i], 1);
            XPRBaddqterm(cobj, y[i], y[j], -2);
            XPRBaddqterm(cobj, y[j], y[j], 1);
        }
    XPRBsetobj(prob, cobj);          /* Set the objective function */

    /**** CONSTRAINTS ****/
    /* All points within given distance of their target location */
    for(i=0;i<N;i++)
    {
        c = XPRBnewctr(prob, XPRBnewname("LimDist_%d",i), XPRB_L);
        XPRBaddqterm(c, x[i], x[i], 1);
        XPRBaddterm(c, x[i], -2*CX[i]);
        XPRBaddterm(c, NULL, -CX[i]*CX[i]);
        XPRBaddqterm(c, y[i], y[i], 1);
        XPRBaddterm(c, y[i], -2*CY[i]);
        XPRBaddterm(c, NULL, -CY[i]*CY[i]);
        XPRBaddterm(c, NULL, R[i]);
    }

    /****SOLVING + OUTPUT****/
    XPRBsetsense(prob, XPRB_MINIM);          /* Sense of optimization */
    XPRBsolve(prob, "");                     /* Solve the problem */

    printf("Solution: %g\n", XPRBgetobjval(prob));
    for(i=0;i<N;i++)
        printf(" %d: %g, %g\n", i, XPRBgetsol(x[i]), XPRBgetsol(y[i]));

    return 0;
}

```

3.5 User error handling

In this section we use a small, infeasible problem to demonstrate how the error handling and all printed messages produced by BCL can be intercepted by the user's program. This is done by defining the corresponding BCL callback functions and changing the error handling flag. If error handling by BCL is disabled, then the definition of the error callback replaces the necessity to check for the return values of the BCL functions called by a program.

User error handling may be required if a BCL program is embedded in some larger application or if the program is run under Windows from an application with windows. In all other cases it will usually be sufficient to use the error handling provided by BCL.

```

#include <stdio.h>
#include <setjmp.h>

```

```

#include <string.h>
#include "xprb.h"

jmp_buf model_failed;          /* Marker for the longjump */

void modinf(XPRBprob prob)
{
    XPRBvar x[3];
    XPRBctr ctr[2], cobj;
    int i;

    for(i=0;i<2;i++)            /* Create two integer variables */
        x[i]=XPRBnewvar(prob, XPRB_UI, XPRBnewname("x_%d",i),0,100);
                                /* Create the constraints:
                                C1: 2x0 + 3x1 >= 41
                                C2:  x0 + 2x1  = 13 */
    ctr[0]=XPRBnewctr(prob,"C1",XPRB_G);
    XPRBaddterm(ctr[0],x[0],2);
    XPRBaddterm(ctr[0],x[1],3);
    XPRBaddterm(ctr[0],NULL,41);

    ctr[1]=XPRBnewctr(prob,"C2",XPRB_E);
    XPRBaddterm(ctr[1],x[0],1);
    XPRBaddterm(ctr[1],x[1],2);
    XPRBaddterm(ctr[1],NULL,13);

    /* Uncomment the following line to cause an error in the model
       that triggers the user error handling: */
    /* x[2]=XPRBnewvar(prob, XPRB_UI, "x_2", 10, 1); */

                                /* Objective: minimize x0+x1 */
    cobj = XPRBnewctr(prob,"OBJ",XPRB_N);
    for(i=0;i<2;i++) XPRBaddterm(cobj, x[i], 1);
    XPRBsetobj(prob,cobj);      /* Select objective function */
    XPRBsetsense(prob,XPRB_MINIM); /* Obj. sense: minimization */

    XPRBprintprob(prob);        /* Print current problem */

    XPRBsolve(prob,"");         /* Solve the LP */
    XPRBprintf(prob,
        "problem status: %d LP status: %d MIP status: %d\n",
        XPRBgetprobstat(prob), XPRBgetlpstat(prob),
        XPRBgetmipstat(prob));

    /* This problem is infeasible, that means the following command
       will fail. It prints a warning if the message level is at
       least 2 */
    XPRBprintf(prob,"Objective: %g\n",XPRBgetobjval(prob));

    for(i=0;i<2;i++)            /* Print solution values */
        XPRBprintf(prob,"%s:%g, ", XPRBgetvarname(x[i]),
            XPRBgetsol(x[i]));
    XPRBprintf(prob,"\n");
}

/**** User error handling function ****/
void XPRB_CC usererror(XPRBprob prob, void *vp, int num,
    int type, const char *t)
{
    printf("BCL error %d: %s\n", num, t);
    if(type==XPRB_ERR) longjmp(model_failed,1);
}

/**** User printing function ****/
void XPRB_CC userprint(XPRBprob prob, void *vp, const char *msg)
{
    static int rtsbefore=1;

    /* Print 'BCL output' whenever a new output line starts,
       otherwise continue to print the current line. */
    if(rtsbefore)
        printf("BCL output: %s", msg);
    else
        printf("%s",msg);
}

```

```

    rtsbefore=msg[strlen(msg)-1]=='\n');
}

int main(int argc, char **argv)
{
    XPRBprob prob;

    XPRBseterrctrl(0);          /* Switch to error handling by the
                                user's program */
    XPRBsetmsglevel(NULL,2);    /* Set the printing flag to
                                printing errors and warnings */
    XPRBdefcbmsg(NULL,userprint, NULL);
                                /* Define the printing callback func. */

    if((prob=XPRBnewprob("ExplInf"))==NULL)
    {
        /* Initialize a new problem in BCL */
        fprintf(stderr,"I cannot create the problem\n");
        return 1;
    }
    else
    {
        if(setjmp(model_failed)) /* Set a marker at this point */
        {
            fprintf(stderr,"I cannot build the problem\n");
            XPRBdelprob(prob);    /* Delete the part of the problem
                                that has been created */
            XPRBdefcberr(prob,NULL, NULL);
                                /* Reset the error callback */
            return 1;
        }
        else
        {
            XPRBdefcberr(prob,usererror, NULL);
                                /* Define the error handling callback */
            modinf(prob);        /* Formulate and solve the problem */
            XPRBdefcberr(prob,NULL, NULL);
                                /* Reset the error callback */
            return 0;
        }
    }
}

```

Since this example defines the printing level and the printing callback function before creating the problem (that is, before BCL is initialized), we pass `NULL` as first argument.

II. BCL library and class reference

Chapter 4

BCL C library functions

A large number of routines are available within the Xpress-MP Builder Component Library, BCL, ranging from simple routines for the creation and solution of problems to sophisticated callback functions and interaction with the Xpress-Optimizer library.

In BCL, references to modeling objects (problem definitions, variables, constraints, sets, and bases) have the following types:

XPRBprob	a problem definition;
XPRBvar	a variable;
XPRBarrvar	a one-dimensional array, with elements of type XPRBvar;
XPRBctr	a constraint;
XPRBcut	a cut;
XPRBsos	a Special Ordered Set (SOS1 or SOS2);
XPRBidxset	an index set;
XPRBbasis	a basis.

4.1 Layout for function descriptions

All functions mentioned in this chapter are described under the following set of headings:

Function name	The description of each routine starts on a new page for the sake of clarity.
Purpose	A short description of the routine and its purpose begins the information section.
Synopsis	A synopsis of the syntax for usage of the routine is provided. 'Optional' arguments and flags may be specified as <code>NULL</code> if not required. Where this possibility exists, it will be described alongside the argument, or in the Further Information at the end of the routine's description.
Arguments	A list of arguments to the routine with a description of possible values for them follows.
Return value	A list of possible return values and their meaning.
Examples	One or two examples are provided which explain certain aspects of the routine's use.
Further information	Additional information not contained elsewhere in the routine's description is provided at the end.
Related topics	Finally a list of related routines and topics is provided for comparison and reference.

XPRBaddarrterm

Purpose

Add multiple linear terms to a constraint.

Synopsis

```
int XPRBaddarrterm(XPRBctr ctr, XPRBarrvar av, double *coeff);
```

Arguments

<code>ctr</code>	Reference to a constraint.
<code>av</code>	Reference to an array of variables.
<code>coeff</code>	Values to be added to the coefficients of the variables in the array (the number of coefficients must correspond to the size of the array of variables).

Return value

0 if function executed successfully, 1 otherwise.

Example

The following adds the expression

$$2*ty1[0] + 13*ty1[1] + 15*ty1[2] + 6*ty1[3] + 8.5*ty1[4]$$

to the constraint `ctrl`.

```
XPRBprob prob;
XPRBctr ctrl;
XPRBarrvar ty1;
double cr[] = {2, 13, 15, 6, 8.5};
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
ctrl = XPRBnewctr(prob, "r1", XPRB_E);
XPRBaddarrterm(ctrl, ty1, cr);
```

Further information

This function adds multiple linear terms to a constraint, the variables coming from array `av` and the corresponding coefficients from `coeff`. If the constraint already has a term with one of the variables, the corresponding value from `coeff` is added to its coefficient.

Related topics

[XPRBaddterm](#), [XPRBdelctr](#), [XPRBdelterm](#), [XPRBnewctr](#).

XPRBaddcutarrterm

Purpose

Add multiple linear terms to a cut.

Synopsis

```
int XPRBaddcutarrterm(XPRBcut cut, XPRBarrvar av, double *coeff);
```

Arguments

<code>cut</code>	Reference to a cut.
<code>av</code>	Reference to an array of variables.
<code>coeff</code>	Values to be added to the coefficients of the variables in the array (the number of coefficients must correspond to the size of the array of variables).

Return value

0 if function executed successfully, 1 otherwise.

Example

Add the term $\sum_{i=0}^4 cr_i \cdot ty1_i$ to the cut `cut1`.

```
XPRBcut cut1;
XPRBarrvar ty1;
double cr[] = {2.0, 13.0, 15.0, 6.0, 8.5};
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
ty1 = XPRBnewarrvar(expl1, 5, XPRB_PL, "arry1", 0, 500);
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
XPRBaddcutarrterm(cut1, ty1, cr);
```

Further information

This function adds multiple linear terms to a cut, the variables coming from array `av` and the corresponding coefficients from `coeff`. If the cut already has a term with one of the variables, the corresponding value from `coeff` is added to its coefficient.

Related topics

[XPRBnewcut](#), [XPRBaddcutterm](#), [XPRBdelcutterm](#).

XPRBaddcuts

Purpose

Add cuts to a problem.

Synopsis

```
int XPRBaddcuts(XPRBprob prob, XPRBcut *cta, int num);
```

Arguments

prob	Reference to a problem.
cta	Array of previously defined cuts.
num	Number of cuts in cta.

Return value

0 if function executed successfully, 1 otherwise.

Example

The example shows how to set up the cut manager node callback to add three previously defined cuts *ca* in node 2 of the MIP search.

```
XPRBcut ca[3];
XPRBprob expl1;

int XPRS_CC usrcme(XPRSprob oprob, void* vd)
{
    int num;
    XPRSgetintattrib(oprob, XPRS_NODES, &num);
    if(num == 2) XPRBaddcuts(expl1, ca, 3);
    return 0;
}

int main(int argc, char **argv)
{
    XPRSprob oprob;
    expl1 = XPRBnewprob("cutexample");
    ... /* Define the problem and the cuts 'ca' */
    XPRBsetcutmode(expl1, 1); /* Enable the cut mode */
    oprob = XPRBgetXPRSprob(expl1); /* Get Optimizer problem */
    XPRSsetcbcutmgr(oprob, usrcme, NULL); /* Set cut mgr. callback */
    XPRBsolve(expl1, "g"); /* Solve the MIP problem */
}
```

Further information

This function adds previously defined cuts to the problem in Xpress-Optimizer. It may only be called from within the Xpress-Optimizer cut manager callback functions. BCL does not check for doubles, that is, if the user defines the same cut twice it will be added twice to the matrix. Cuts added at a node during the branch and bound search remain valid for all child nodes but are removed at all other nodes.

Related topics

[XPRBnewcut](#), [XPRBdelcut](#), [XPRBsetcutmode](#).

XPRBaddcutterm

Purpose

Add a term to a cut.

Synopsis

```
int XPRBaddcutterm(XPRBcut cut, XPRBvar var, double coeff);
```

Arguments

cut Reference to a cut as resulting from [XPRBnewcut](#).
var Reference to a variable, may be `NULL`.
coeff Value to be added to the coefficient of the variable `var`.

Return value

0 if function executed successfully, 1 otherwise.

Example

Add the term $5.4 \cdot x_1$ to the cut *cut1*.

```
XPRBcut cut1;  
XPRBvar x1;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
x1 = XPRBnewvar(expl1, XPRB_UI, "abc3", 0, 100);  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
XPRBaddcutterm(cut1, x1, 5.4);
```

Further information

This function adds a new term to a cut, comprising the variable `var` with coefficient `coeff`. If the cut already has a term with variable `var`, `coeff` is added to its coefficient. If `var` is set to `NULL`, the value `coeff` is added to the right hand side of the cut.

Related topics

[XPRBnewcut](#), [XPRBaddcutarrterm](#) [XPRBdelcutterm](#), [XPRBsetcutterm](#).

XPRBaddidxel

Purpose

Add an index to an index set.

Synopsis

```
int XPRBaddidxel(XPRBidxset idx, const char *name);
```

Arguments

<code>idx</code>	A BCL index set.
<code>name</code>	Name of the index to be added to the set.

Return value

Sequence number of the index within the set, -1 in case of an error.

Example

The following defines an index set with space for 100 entries, adds an index to the set and then retrieves its sequence number.

```
XPRBprob prob;  
XPRBidxset iset;  
int val;  
...  
iset = XPRBnewidxset(prob, "Set", 100);  
val = XPRBaddidxel(iset, "first");
```

Further information

This function adds an index entry to a previously defined index set. The new element is only added to the set if no identical index already exists. Both in the case of a new index entry and an existing one, the function returns the sequence number of the index in the index set. Note that, according to the usual C convention, the numbering of index elements starts with 0.

Related topics

[XPRBgetidxel](#), [XPRBnewidxset](#).

XPRBaddqterm

Purpose

Add a quadratic term to a constraint.

Synopsis

```
int XPRBaddqterm(XPRBctr ctr, XPRBvar var1, XPRBvar var2,
                 double coeff);
```

Arguments

ctr Reference to a constraint.
var1 Reference to a variable.
var2 Reference to a variable (not necessarily different).
coeff Value to be added to the coefficient of the term $\text{var1} * \text{var2}$.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following example adds the term $-2*x2*x4$ to the constraint `ctrl`:

```
XPRBctr ctrl;  
XPRBvar x2,x4;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_L);  
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);  
x4 = XPRBnewvar(prob, XPRB_PL, "abc5", 0, XPRB_INFINITY);  
XPRBaddqterm(ctrl, x2, x4, -2);
```

Further information

This function adds a new quadratic term to a constraint, comprising the product of the variables `var1` and `var2` with coefficient `coeff`. If the constraint already has a term with variables `var1` and `var2`, `coeff` is added to its coefficient.

Related topics

[XPRBdelqterm](#), [XPRBsetqterm](#).

XPRBaddsosarrel

Purpose

Add multiple elements to a SOS.

Synopsis

```
int XPRBaddsosarrel(XPRBsos sos, XPRBarrvar av, double *weight);
```

Arguments

<code>sos</code>	A SOS of type 1 or 2.
<code>av</code>	An array of variables.
<code>weight</code>	An array of weight coefficients. The number of weights must correspond to the size of the array of variables.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following adds an array `ty1` with weights `cr` to the SOS `set1`.

```
XPRBprob prob;  
XPRBsos set1;  
XPRBarrvar ty1;  
double cr[] = {2, 13, 15, 6, 8.5};  
...  
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
XPRBaddsosarrel(set1, ty1, cr);
```

Further information

This function adds an array of variables and their corresponding weights (reference values) to a SOS. If a variable is already contained in the set, the indicated value is added to its weight. Note that all weight coefficients must be different from 0.

Related topics

[XPRBaddsosel](#), [XPRBdelsos](#), [XPRBdelsosel](#), [XPRBnewsos](#).

XPRBaddsosel

Purpose

Add an element to a SOS.

Synopsis

```
int XPRBaddsosel(XPRBsos sos, XPRBvar var, double weight);
```

Arguments

sos	A SOS of type 1 or 2.
var	Reference to a variable.
weight	The corresponding weight or reference value.

Return value

0 if function executed successfully, 1 otherwise

Example

```
XPRBprob prob;  
XPRBsos set1;  
XPRBvar x2;  
...  
x2 = XPRBnewvar(prob, XPRB_PL, " abc1", 0 ,X PRB_INFINITY);  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
XPRBaddsosel(set1, x2, 9);
```

This adds a variable `x2` with weight 9 to the SOS `set1`.

Further information

This function adds a single variable and its weight coefficient to a Special Ordered Set. If the variable is already contained in the set, the indicated value is added to its weight. Note that weight coefficients must be different from 0.

Related topics

[XPRBaddsosarrel](#), [XPRBdelsos](#), [XPRBdelsosel](#), [XPRBnewsos](#).

XPRBaddterm

Purpose

Add a linear term to a constraint.

Synopsis

```
int XPRBaddterm(XPRBctr ctr, XPRBvar var, double coeff);
```

Arguments

ctr BCL reference to a constraint, resulting from [XPRBnewctr](#).
var BCL reference to a variable. May be `NULL` if not required.
coeff Amount to be added to the coefficient of the variable `var`.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBctr ctrl;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);  
XPRBaddterm(ctrl, x1, 5.4);
```

This adds the term $5.4 \times x1$ to the constraint `ctrl`.

Further information

This function adds a new linear term to a constraint, comprising the variable `var` with coefficient `coeff`. If the constraint already has a term with variable `var`, `coeff` is added to its coefficient. If `var` is set to `NULL`, the value `coeff` is added to the right hand side of the constraint.

Related topics

[XPRBaddarrterm](#), [XPRBaddqterm](#), [XPRBdelctr](#), [XPRBdelterm](#), [XPRBnewctr](#), [XPRBsetterm](#).

XPRBapparrvarel

Purpose

Add an entry to a variable array.

Synopsis

```
int XPRBapparrvarel(XPRBarrvar av, XPRBvar var);
```

Arguments

<code>av</code>	BCL reference to an array.
<code>var</code>	The variable to be added.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following inserts the variable `x1` in the first free position of the array `av2`.

```
XPRBprob prob;  
XPRBarrvar av2;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
av2 = XPRBstartarrvar(prob, 5, "arr2");  
XPRBapparrvarel(av2, x1);
```

Further information

This function inserts a variable in the first available position within an array.

Related topics

[XPRBdelarrvar](#), [XPRBendarrvar](#), [XPRBnewarrvar](#), [XPRBsetarrvarel](#), [XPRBstartarrvar](#).

XPRBcleardir

Purpose

Delete all directives.

Synopsis

```
int XPRBcleardir(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBcleardir(expl2);
```

This deletes all directives for the current problem, `expl2`.

Related topics

[XPRBsetvardir](#), [XPRBsetsosdir](#).

XPRBdefcbdelvar

Purpose

Callback for interface update at deletion of variables.

Synopsis

```
int XPRBpdefcbdelvar(XPRBprob prob,
                    void (XPRB_CC *delinter)(XPRBprob eprob, void *evp,
                    XPRBvar var, void *link), void *vp);
```

Arguments

prob	Reference to a problem.
delinter	User variable interface update function
eprob	Problem from which the callback is called
evp	Empty pointer for passing additional information
var	Reference to a BCL variable
link	Pointer to an interface object
vp	Empty pointer for the user to pass additional information

Return value

0 if function executed successfully, 1 otherwise.

Example

Define the variable interface callback function:

```
XPRBprob prob;
...
void mydelinter(XPRBprob prob, void *vp, XPRBvar var, void *adr)
{
    printf("Deleted: %s", XPRBgetvarname(var));
}
...
XPRBdefcbdelvar(prob, mydelinter, NULL);
```

Further information

This function defines a callback function that is called at the deletion of any variable that is used in an interface to an external program, (that means, if the interface pointer of the variable is different from `NULL`).

Related topics

[XPRBgetvarlink](#), [XPRBsetvarlink](#).

XPRBdefcberr

Purpose

Callback for user error handling.

Synopsis

```
int XPRBdefcberr(XPRBprob prob,
                void (XPRB_CC *usererr)(XPRBprob my_prob, void *my_object,
                int errnum, int type, const char *errtext), void *object);
```

Arguments

prob	Reference to a problem.
usererr	The user's error handling function.
my_prob	Problem pointer passed to the callback function.
my_object	User-defined object passed to the callback function.
errnum	The error number.
type	Type of the error. This will be one of: XPRB_ERR fatal error; XPRB_WAR warning.
errtext	Text of the error message.
object	User-defined object to be passed to the callback function.

Return value

0 if function executed successfully, 1 otherwise.

Example

In this example a function is defined for displaying errors and exiting if they are suitably severe. This function is then set as the error-handling callback.

```
XPRBprob prob;
...
void myerr(XPRBprob my_prob, void *my_object, int num, int type,
          const char *t)
{
    printf("BCL error %d: %s\n", num, t);
    if(type == XPRB_ERR) exit(0);
}
...
XPRBdefcberr(prob, myerror, NULL);
```

Further information

1. This function defines the error handling callback that returns the error number and text of error messages and warnings produced by BCL for a given problem. A list of BCL error messages with some explanations can be found in the Appendix A of this manual. If printing of error or warning messages is enabled (see [XPRBsetmsglevel](#)) these are printed after the call to this function.
2. It is recommended to define this callback function if the error handling by BCL is disabled (for instance in BCL programs integrated into larger applications or in BCL programs executed under Windows). Alternatively it is of course possible to test the return values of all BCL functions. However, the callback provides more detailed information about the type of error that has occurred.
3. This function may be used before any problems have been created and even before BCL has been initialized (with first argument NULL). In this case the error handling function set by this callback applies to all problems that are created subsequently.

Related topics

[XPRBdefcbmsg](#), [XPRBgetversion](#), [XPRBseterrctrl](#).

XPRBdefcbmsg

Purpose

Callback for printed output.

Synopsis

```
int XPRBdefcbmsg(XPRBprob prob,
                 void (XPRB_CC *userprint)(XPRBprob my_prob, void *my_object,
                 const char *msgtext), void *object);
```

Arguments

prob	Reference to a problem.
userprint	A user message handling function.
my_prob	Problem pointer passed to the callback function.
my_object	User-defined object passed to the callback function.
msgtext	The message text.
object	User-defined object to be passed to the callback function.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following defines a print function and then sets it as a callback.

```
XPRBprob prob;
...
void myprint(XPRBprob prob, void *my_object, const char *msg);
{
    printf("BCL output: %s\n", msg);
}
...
XPRBdefcbmsg(prob, myprint, NULL);
```

Further information

1. This function defines a callback function that returns any messages enabled by the setting of [XPRBsetmsglevel](#), including warnings and error messages, any other output produced by BCL, and any messages from the Optimizer library. Independent of the message printing settings, this callback also returns output printed by the user's program with function [XPRBprintf](#). If this callback is not defined by the user, any program output is printed to standard output with the exception of warnings and error messages which are printed to the standard error output channel.
2. This function may be used before any problems have been created and even before BCL has been initialized (with first argument NULL). In this case the printing function set by this callback applies to all problems that are created subsequently.
3. A BCL program must *not* define the message callback [XPRSsetcbmessage](#) of Xpress-Optimizer (however, all other logging callbacks of the Optimizer may be used).

Related topics

[XPRBdefcberr](#), [XPRBsetmsglevel](#).

XPRBdelarrvar

Purpose

Delete a variable array.

Synopsis

```
int XPRBdelarrvar(XPRBarrvar av);
```

Argument

`av` BCL reference to an array in the model.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBarrvar av2;  
...  
av2 = XPRBstartarrvar(prob, 5, "arr2");  
XPRBendarrvar(av2);  
XPRBdelarrvar(av2);
```

This deletes the array `av2`, although not any variables that may have been added to it.

Further information

This function deletes the reference to an array. Arrays may be used as auxiliary constructs for defining constraints. This means it may not be necessary to keep them. If an array is only used in the model, it can be deleted by a call to this function, thus freeing the corresponding memory allocated to it. The variables belonging to the array are not deleted by this function if the array has been created with [XPRBstartarrvar](#).

Related topics

[XPRBapparrvarel](#), [XPRBendarrvar](#), [XPRBnewarrvar](#), [XPRBsetarrvarel](#),
[XPRBstartarrvar](#).

XPRBdelbasis

Purpose

Delete a previously saved basis.

Synopsis

```
void XPRBdelbasis(XPRBbasis basis);
```

Argument

`basis` Reference to a previously saved basis.

Example

The following code demonstrates saving a basis prior to some matrix changes. Subsequently the old basis is reloaded and the redundant saved basis deleted.

```
XPRBprob expl2;
XPRBbasis basis;
expl2 = XPRBnewprob("example2");
...
XPRBsolve(expl2, "1");
basis = XPRBsavebasis(expl2);
...
XPRBloadmat(expl2);
XPRBloadbasis(basis);
XPRBdelbasis(basis);
XPRBsolve(expl2, "1");
```

Further information

This function deletes a basis that has been saved using function [XPRBsavebasis](#). Typically, the reference to a basis should be deleted if it is not used any more.

Related topics

[XPRBloadbasis](#), [XPRBsavebasis](#).

XPRBdelctr

Purpose

Delete a constraint.

Synopsis

```
int XPRBdelctr(XPRBctr ctr);
```

Argument

`ctr` BCL reference to a constraint.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBctr ctrl;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);  
XPRBdelctr(ctrl);
```

This deletes the constraint `ctrl`.

Further information

Delete a constraint from the given problem. If this constraint has previously been selected as the objective function (using function [XPRBsetobj](#)), the objective will be set to `NULL`.

Related topics

[XPRBnewctr](#).

XPRBdelcut

Purpose

Delete a cut definition.

Synopsis

```
int XPRBdelcut(XPRBcut cut);
```

Argument

`cut` Reference to a cut.

Return value

0 if function executed successfully, 1 otherwise.

Example

The example shows how to delete cut *cut1*.

```
XPRBcut cut1;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
XPRBdelcut(cut1);
```

Further information

This function deletes the definition of a cut in BCL, but *not* the cut itself if it has already been added to the problem held in Xpress-Optimizer (using function [XPRBaddcuts](#)).

Related topics

[XPRBnewcut](#), [XPRBaddcuts](#).

XPRBdelcutterm

Purpose

Delete a term from a cut.

Synopsis

```
int XPRBdelcutterm(XPRBcut cut, XPRBvar var);
```

Arguments

<code>cut</code>	Reference to a cut as resulting from XPRBnewcut .
<code>var</code>	Reference to a variable in the cut.

Return value

0 if function executed successfully, 1 otherwise.

Example

Add the term $5.4 \cdot x_1$ to the cut *cut1* and then delete it.

```
XPRBcut cut1;
XPRBvar x1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
x1 = XPRBnewvar(expl1, XPRB_UI, "abc3", 0, 100);
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
XPRBaddcutterm(cut1, x1, 5.4);
XPRBdelcutterm(cut1, x1);
```

Further information

This function removes a variable term from a cut. The constant term (right hand side value) is changed/reset with function [XPRBsetcutterm](#).

Related topics

[XPRBnewcut](#), [XPRBaddcutarrterm](#) [XPRBaddcutterm](#), [XPRBsetcutterm](#).

XPRBdelprob

Purpose

Delete a problem.

Synopsis

```
int XPRBdelprob(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

In this example, the problem `expl2` is deleted.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
XPRBdelprob(expl2);
```

Further information

This function deletes the given problem in BCL, and the corresponding problem in the Optimizer. It also deletes any remaining working files associated with this problem. All parameter settings remain valid after deleting a problem. If the user does not wish to delete a problem but wants to free some resources used for storing solution information he may call [XPRBresetprob](#).

Related topics

[XPRBnewprob](#), [XPRBresetprob](#).

XPRBdelqterm

Purpose

Delete a quadratic term from a constraint.

Synopsis

```
int XPRBdelqterm(XPRBctr ctr, XPRBvar var1, XPRBvar var2);
```

Arguments

`ctr` Reference to a constraint.
`var1` Reference to a variable.
`var2` Reference to a variable (not necessarily different).

Return value

0 if function executed successfully, 1 otherwise.

Example

The following example first adds the term $5.2 \times x_2 \times x_2$ to the constraint `ctrl` and then deletes this term from the constraint:

```
XPRBctr ctrl;  
XPRBvar x2,x4;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_L);  
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);  
XPRBaddqterm(ctrl, x2, x2, 5.2);  
XPRBdelqterm(ctrl, x2, x2);
```

Further information

This function deletes a quadratic term from a constraint, comprising the product of the variables `var1` and `var2`.

Related topics

[XPRBaddqterm](#), [XPRBsetqterm](#).

XPRBdelsos

Purpose

Delete a SOS.

Synopsis

```
int XPRBdelsos(XPRBsos sos);
```

Argument

`sos` Reference to a previously defined SOS of type 1 or 2.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following deletes the SOS `set1`.

```
XPRBprob prob;  
XPRBsos set1;  
...  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
XPRBdelsos(set1);
```

Further information

This function deletes a SOS without deleting the variables it consists of.

Related topics

[XPRBaddsosarrel](#), [XPRBaddsosel](#), [XPRBdelsosel](#), [XPRBnewsos](#).

XPRBdelssel

Purpose

Delete an element from a SOS.

Synopsis

```
int XPRBdelssel(XPRBsos sos, XPRBvar var);
```

Arguments

<code>sos</code>	A SOS of type 1 or 2.
<code>var</code>	Reference to a variable.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following removes the variable `x2` from the SOS `set1`.

```
XPRBprob prob;  
XPRBsos set1;  
XPRBvar x2;  
...  
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
XPRBaddssel(set1, x2, 9.0);  
XPRBdelssel(set1, x2);
```

Further information

This function removes a variable from a Special Ordered Set.

Related topics

[XPRBaddsarrel](#), [XPRBaddssel](#), [XPRBdelsos](#), [XPRBnewsos](#).

XPRBdelterm

Purpose

Delete a linear term from a constraint.

Synopsis

```
int XPRBdelterm(XPRBctr ctr, XPRBvar var);
```

Arguments

<code>ctr</code>	BCL reference to a previously created constraint.
<code>var</code>	BCL reference to a variable.

Return value

0 if function executed successfully, 1 otherwise.

Example

This code deletes the variable `x1` from the constraint.

```
XPRBprob prob;  
XPRBctr ctrl;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);  
XPRBaddterm(ctrl, x1, 5.4);  
XPRBdelterm(ctrl, x1);
```

Further information

This function deletes a linear term from the given constraint.

Related topics

[XPRBaddarrterm](#), [XPRBaddterm](#), [XPRBdelctr](#), [XPRBnewctr](#), [XPRBsetterm](#).

XPRBendarrvar

Purpose

End the definition of a variable array.

Synopsis

```
int XPRBendarrvar(XPRBarrvar av);
```

Argument

`av` BCL reference to an array.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBarrvar av2;  
...  
av2 = XPRBstartarrvar(prob, 5, "arr2");  
XPRBendarrvar(av2);
```

This terminates the definition of the array `av2`.

Further information

This function terminates the definition of the array. As the reference to the array is required by this function in common with all other functions referring to the incremental definition of arrays it is possible to define several arrays at a time.

Related topics

[XPRBdelarrvar](#), [XPRBnewarrvar](#), [XPRBstartarrvar](#).

XPRBexportprob

Purpose

Print problem matrix to a file.

Synopsis

```
int XPRBexportprob(XPRBprob prob, int format, char *filename);
```

Arguments

prob	Reference to a problem.
format	The matrix output file format, which must be one of: XPRB_LP LP file format (default); XPRB_MPS MPS file format.
filename	Name of the output file, without extension.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
XPRBexportprob(expl2, XPRB_MPS, "ex2");
```

This prints the problem in MPS format to the file `ex2.mat`.

Further information

1. This function prints the matrix to a file with an extended LP or extended MPS format. LP files receive the extension `.lp` and MPS files receive the extension `.mat`. This function is not available in the student version.
2. When exporting matrices semi-continuous and semi-continuous integer variables are preprocessed: if a lower bound value greater than 0 is given, then the variable is treated like a continuous (resp. integer) variable.

Related topics

[XPRBprintprob](#), [XPRBprintf](#).

XPRBfinish, XPRBfree

Purpose

Terminate BCL and release system resources.

Synopsis

```
int XPRBfinish(void);  
int XPRBfree(void);
```

Return value

0 if function executed successfully, 1 otherwise.

Example

The following tidies up at the end of a BCL session:

```
XPRBprob prob;  
prob = XPRBnewprob(NULL);  
...  
XPRBdelprob(prob);  
XPRBfinish();
```

Further information

Importantly, `XPRBfinish` does not free memory associated with problems. These should all be removed using the `XPRBdelprob` function. When running programs that are mainly based on BCL there is no need to call this function since system resources are freed at the end of the program. To the contrary, it may be interesting to be able to reset and free resources if a BCL program is embedded into some larger application that continues to work after the BCL part has finished. If the user does not wish to delete a problem or terminate BCL but wants to free some resources used for storing solution information he may call `XPRBresetprob`. Note that `XPRBfinish` also terminates Xpress-Optimizer if it has been started through BCL. If the Optimizer has been started with an explicit call to `XPRSinit` before BCL has been started, then it is not terminated by `XPRBfinish`.

Related topics

`XPRBdelprob`, `XPRBresetprob`, `XPRBinit`.

XPRBfixvar

Purpose

Fix a variable.

Synopsis

```
int XPRBfixvar(XPRBvar var, double val);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>val</code>	The value to which the variable is to be fixed.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code sets the value of variable `x1` to 20.

```
XPRBprob prob;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 1, 100);  
XPRBfixvar(x1, 20.0);
```

Further information

This function fixes a variable to the given value. It replaces calls to [XPRBsetub](#) and [XPRBsetlb](#). The value `val` may lie outside the original bounds of the variable.

Related topics

[XPRBgetbounds](#), [XPRBgetlim](#), [XPRBsetlb](#), [XPRBsetlim](#), [XPRBsetub](#).

XPRBgetact

Purpose

Get activity value for a constraint.

Synopsis

```
double XPRBgetact(XPRBctr ctr);
```

Argument

`ctr` Reference to a constraint.

Return value

Activity value for the constraint, 0 in case of an error.

Example

```
XPRBprob expl2;
XPRBctr ctr2;
XPRBarrvar ty1;
double act
...
expl2 = XPRBnewprob("example2");
ty1 = XPRBnewarrvar(expl2, 5, XPRB_PL, "arry1", 0, 500);
ctr2 = XPRBnewsum(expl2, "r2", ty1, XPRB_E, 9);
XPRBsolve(expl2, "1");
act = XPRBgetact(ctr2);
```

This obtains the activity value for the constraint `ctr2`.

Further information

This function returns the activity value for a constraint. It may be used with constraints that are not part of the problem (in particular, constraints without relational operators, that is, constraints of type `XPRB_N`). In this case the function returns the evaluation of the constraint terms involving variables that are in the problem. Otherwise, the constraint activity is calculated as *activity* = *RHS* – *slack*.

If this function is called after completion of a global search and an integer solution has been found (that is, if function `XPRBgetmipstat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value corresponding to the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the activity value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBsync` with the flag `XPRB_XPRS_SOL`.

Related topics

`XPRBgetdual`, `XPRBgetobjval`, `XPRBgetrcost`, `XPRBgetslack`, `XPRBgetsol`, `XPRBsync`.

XPRBgetarrvarname

Purpose

Get the name of an array of variables.

Synopsis

```
const char *XPRBgetarrvarname(XPRBarrvar av);
```

Argument

`av` BCL reference to an array of variables.

Return value

Name of the array if function executed successfully, `NULL` otherwise.

Example

```
XPRBprob prob;  
XPRBarrvar ty1;  
...  
ty1 = XPRBnewarrvar(prob, 10, XPRB_PL, "arry1", 0, 500);  
printf("%s\n", XPRBgetarrvarname(ty1));
```

This prints the output `arry1`, the array variable name.

Further information

This function returns the name of an array of variables. If the name was not set by the user, this is a default name generated by BCL.

Related topics

[XPRBdelarrvar](#), [XPRBgetarrvarsize](#), [XPRBnewarrvar](#).

XPRBgetarrvarsize

Purpose

Get the size of an array of variables.

Synopsis

```
int XPRBgetarrvarsize(XPRBarrvar av);
```

Argument

`av` BCL reference to an array of variables.

Return value

Size (= number of variables) of the array, or -1 in case of an error.

Example

```
XPRBprob prob;  
XPRBarrvar ty1;  
int tsize;  
...  
ty1 = XPRBnewarrvar(prob, 10, XPRB_PL, "arry1", 0, 500);  
tsize = XPRBgetarrvarsize(ty1);
```

This gets the size of the array `ty1`.

Further information

This function returns the size (*i.e.* the number of elements) of an array of variables. If the variables have been added incrementally the returned value may be smaller than the maximum size given at the creation of the array. The returned size represents the number of variables that have actually been added to the array.

Related topics

[XPRBdelarrvar](#), [XPRBgetarrvarname](#), [XPRBnewarrvar](#).

XPRBgetbounds

Purpose

Get the bounds on a variable.

Synopsis

```
int XPRBgetbounds(XPRBvar var, double *bdl, double *bdu);
```

Arguments

var	BCL reference to a variable.
bdl	Lower bound value. May be <code>NULL</code> if not required.
bdu	Upper bound value. May be <code>NULL</code> if not required.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBvar x1;  
double ubound;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
XPRBgetbounds(x1, NULL, &ubound);
```

This retrieves the upper bound of the variable `x1`.

Further information

This function returns the currently defined bounds on a variable. If `bdl` or `bdu` is set to `NULL`, no value is returned into the corresponding argument.

Related topics

[XPRBfixvar](#), [XPRBgetlim](#), [XPRBsetlb](#), [XPRBsetlim](#), [XPRBsetub](#).

XPRBgetbyname

Purpose

Retrieve an object by its name.

Synopsis

```
void *XPRBgetbyname(XPRBprob prob, const char *name, int type);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>name</code>	The name of the object.
<code>type</code>	The type of the object sought. This is one of: XPRB_VAR a BCL variable; XPRB_ARR a BCL array of variables; XPRB_CTR a BCL constraint; XPRB_SOS a BCL SOS; XPRB_IDX a BCL index set.

Return value

Reference to a BCL object of the indicated type if function executed successfully, `NULL` if object not found or in case of an error.

Example

This example finds the variable with the name `abc3`.

```
XPRBprob prob;  
XPRBvar x1;  
...  
x1 = XPRBgetbyname(prob, "abc3", XPRB_VAR);
```

Further information

The function returns the reference to an object of the indicated type or `NULL`. The same name may be used for objects of different types within one problem definition. This function can only be used if the names dictionary is enabled (functions `XPRBsetdictionarysize`).

Related topics

`XPRBsetdictionarysize`, `XPRBnewname`.

XPRBgetcolnum

Purpose

Get the column number for a variable.

Synopsis

```
int XPRBgetcolnum(XPRBvar var);
```

Argument

`var` BCL reference to a variable.

Return value

Column number (non-negative value), or a negative value.

Example

```
XPRBprob expl2;  
XPRBvar x1;  
int vindex;  
...  
expl2 = XPRBnewprob("example2");  
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 0, 100);  
vindex = XPRBgetcolnum(x1);
```

This gets the column number for variable `x1`.

Further information

This function returns the column number of a variable in the matrix currently loaded in the Xpress-Optimizer. If the variable is not part of the matrix, or if the matrix has not yet been generated, the function returns a negative value. To check whether the matrix has been generated, use function [XPRBgetprobat](#). The counting of column numbers starts with 0.

Related topics

[XPRBgetvarname](#), [XPRBgetvartype](#).

XPRBgetctrname

Purpose

Get the name of a constraint.

Synopsis

```
const char *XPRBgetctrname(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

Name of the constraint if function executed successfully, `NULL` otherwise

Example

```
XPRBprob expl2;
XPRBctr ctr1;
...
expl2 = XPRBnewprob("example2");
ctr1 = XPRBnewctr(expl2, "r1", XPRB_E);
printf("%s\n", XPRBgetctrname(ctr1));
```

This prints "r1" as its output.

Further information

This function returns the name of a constraint. If the user has not defined a name the default name generated by BCL is returned.

Related topics

[XPRBgetctrtype](#), [XPRBnewctr](#).

XPRBgetctrng

Purpose

Get ranging information for a constraint.

Synopsis

```
double XPRBgetctrng(XPRBctr ctr, int rngtype);
```

Arguments

<code>ctr</code>	Reference to a previously created constraint.
<code>rngtype</code>	The type of ranging information sought. This is one of: XPRB_UPACT upper activity; XPRB_LOACT lower activity; XPRB_UUP upper unit cost; XPRB_UDN lower unit cost.

Return value

Ranging information of the required type.

Example

The following returns the upper activity value of the constraint `ctr1`.

```
XPRBprob expl2;  
XPRBctr ctr1;  
double upact;  
expl2 = XPRBnewprob("example2");  
ctr1 = XPRBnewctr(expl2, "r1", XPRB_E);  
...  
XPRBsolve(expl2, "l");  
upact = XPRBgetctrng(ctr1, XPRB_UPACT);
```

Further information

This method can only be used after solving an LP problem. Ranging information for MIP problems can be obtained by fixing all discrete variables to their solution values and re-solving the resulting LP problem.

Related topics

[XPRBnewctr](#), [XPRBgetvarrng](#).

XPRBgetctrtype

Purpose

Get the row type of a constraint.

Synopsis

```
int XPRBgetctrtype(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

`XPRB_L` 'less than or equal to' inequality;
`XPRB_G` 'greater than or equal to' inequality;
`XPRB_E` equality;
`XPRB_N` a non-binding row (objective function);
`XPRB_R` a range constraint;
`-1` an error has occurred.

Example

The following returns the type of the constraint `ctrl`.

```
XPRBprob expl2;  
XPRBctr ctrl;  
char rtype;  
...  
expl2 = XPRBnewprob("example2");  
ctrl = XPRBnewctr(expl2, "r1", XPRB_E);  
rtype = XPRBgetctrtype(ctrl);
```

Further information

The function returns the constraint type if successful, and -1 in case of an error.

Related topics

[XPRBgetctrname](#), [XPRBnewctr](#), [XPRBsetctrtype](#).

XPRBgetcutid

Purpose

Get the classification or identification number of a cut.

Synopsis

```
int XPRBgetcutid(XPRBcut cut);
```

Argument

`cut` Reference to a previously created cut.

Return value

Classification or identification number.

Example

Get the classification or identification number of the cut *cut1*.

```
XPRBcut cut1;  
int cid;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
cid = XPRBgetcutid(cut1);
```

Further information

This function returns the classification or identification number of a previously defined cut.

Related topics

[XPRBnewcut](#), [XPRBgetcuttype](#), [XPRBgetcutrhs](#), [XPRBsetcutid](#).

XPRBgetcutrhs

Purpose

Get the RHS value of a cut.

Synopsis

```
double XPRBgetcutrhs(XPRBcut cut);
```

Argument

`cut` Reference to a previously created cut.

Return value

Right hand side (RHS) value (default 0).

Example

Get the RHS value of the cut *cut1*.

```
XPRBcut cut1;
double rhs;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
rhs = XPRBgetcutrhs(cut1);
```

Further information

This function returns the RHS value (= constant term) of a previously defined cut. The default RHS value is 0.

Related topics

[XPRBnewcut](#), [XPRBaddcutterm](#), [XPRBgetcutid](#), [XPRBgetcuttype](#).

XPRBgetcuttype

Purpose

Get the type of a cut.

Synopsis

```
int XPRBgetcuttype(XPRBcut cut);
```

Argument

`cut` Reference to a previously created cut.

Return value

`XPRB_L` \leq (inequality)
`XPRB_G` \geq (inequality)
`XPRB_E` = (equation)
`-1` An error has occurred,

Example

Get the type of *cut1*.

```
XPRBcut cut1;  
int rtype;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
rtype = XPRBgetcuttype(cut1);
```

Further information

This function returns the type of the given cut.

Related topics

[XPRBnewcut](#), [XPRBgetcutid](#), [XPRBgetcutrhs](#), [XPRBsetcuttype](#).

XPRBgetdual

Purpose

Get dual value.

Synopsis

```
double XPRBgetdual(XPRBctr ctr);
```

Argument

`ctr` Reference to a constraint.

Return value

Dual value for the constraint, 0 in case of an error.

Example

```
XPRBprob expl2;
XPRBctr ctr2;
XPRBarrvar ty1;
double dval;
...
expl2 = XPRBnewprob("example2");
ty1 = XPRBnewarrvar(expl2, 5, XPRB_PL, "arry1", 0, 500);
ctr2 = XPRBnewsum(expl2, "r2", ty1, XPRB_E, 9);
XPRBsolve(expl2, "1");
dval = XPRBgetdual(ctr2);
```

This obtains the dual value for the constraint `ctr2`.

Further information

This function returns the dual value for a constraint. The user may wish to test first whether this constraint is part of the problem, for instance by checking that the row number is non-negative.

If this function is called after completion of a global search and an integer solution has been found (that is, if function [XPRBgetmipstat](#) returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the dual value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to [XPRBsync](#) with the flag `XPRB_XPRS_SOL`.

Related topics

[XPRBgetactivity](#), [XPRBgetobjval](#), [XPRBgetrcost](#), [XPRBgetslack](#), [XPRBgetsol](#), [XPRBsync](#).

XPRBgetidxel

Purpose

Get the index number of an index.

Synopsis

```
int XPRBgetidxel(XPRBidxset idx, char *name);
```

Arguments

<code>idx</code>	A BCL set name
<code>name</code>	Name of an index in the set.

Return value

Sequence number of the index in the set, or -1 if not contained.

Example

```
XPRBprob prob;  
XPRBidxset iset;  
int val;  
  
...  
iset = XPRBnewidxset(prob, "Set", 100);  
XPRBaddidxel(iset, "first");  
val = XPRBgetidxel(iset, "first");
```

This defines an index set , `iset`, with space for 100 entries, adds an index, `first`, to the set and subsequently retrieves its sequence number.

Further information

An index element can be accessed either by its name or by its sequence number. This function returns the sequence number of an index given its name.

Related topics

[XPRBaddidxel](#), [XPRBnewidxset](#).

XPRBgetidxelname

Purpose

Get the name of an index.

Synopsis

```
const char *XPRBgetidxelname(XPRBidxset idx, int i);
```

Arguments

<code>idx</code>	A BCL index set.
<code>i</code>	Index number.

Return value

Name of the i^{th} element in the set if function executed successfully, `NULL` otherwise.

Example

```
XPRBprob prob;  
XPRBidxset iset;  
const char *name;  
  
...  
iset = XPRBnewidxset(prob, "Set", 100);  
name = XPRBgetidxelname(iset, 0);
```

This defines an index set, `iset`, with space for 100 entries and retrieves the name of the index set element with sequence number 0.

Further information

An index element can be accessed either by its name or by its sequence number. This function returns the name of an index set element given its sequence number.

Related topics

[XPRBaddidxel](#), [XPRBgetidxsetname](#), [XPRBgetidxel](#), [XPRBnewidxset](#).

XPRBgetidxsetname

Purpose

Get the name of an index set.

Synopsis

```
const char *XPRBgetidxsetname(XPRBidxset idx);
```

Argument

`idx` A BCL index set.

Return value

Name of the index set if function executed successfully, `NULL` otherwise.

Example

The following defines an index set, `iset`, with space for 100 entries and then retrieves its name.

```
XPRBprob prob;  
XPRBidxset iset;  
const char *name;  
  
...  
iset = XPRBnewidxset(prob, "Set", 100);  
name = XPRBgetidxsetname(iset);
```

Further information

This function returns the name of an index set.

Related topics

[XPRBgetidxelname](#), [XPRBgetidxsetsize](#), [XPRBnewidxset](#).

XPRBgetidxsetsize

Purpose

Get the size of an index set.

Synopsis

```
int XPRBgetidxsetsize(XPRBidxset idx);
```

Argument

`idx` A BCL index set.

Return value

Size (= number of elements) of the set, -1 in case of an error.

Example

The following defines an index set with space for 100 elements and then retrieves its size.

```
XPRBprob prob;  
XPRBidxset iset;  
int size;  
...  
iset = XPRBnewidxset(prob, "Set", 100);  
size = XPRBgetidxsetsize(iset);
```

Further information

This function returns the current number of elements in an index set. This value does not necessarily correspond to the size specified at the creation of the set. The returned value may be smaller if fewer elements than the originally reserved number have been added, or larger if more elements have been added. (In the latter case, the size of the set is automatically increased.)

Related topics

[XPRBaddidxel](#), [XPRBgetidxsetname](#), [XPRBnewidxset](#).

XPRBgetiis

Purpose

Get the variables and constraints of an IIS.

Synopsis

```
int XPRBgetiis(XPRBprob prob, XPRBvar **arrvar, int *numv, XPRBctr
               **arrctr, int *numc, int numiis);
```

Arguments

prob	Reference to a problem.
arrvar	Reference to a table of BCL variables (may be <code>NULL</code>).
numv	Reference to an integer that gets assigned the number of variables returned by the function (may be <code>NULL</code>).
arrctr	Reference to a table of BCL constraints (may be <code>NULL</code>).
numc	Reference to an integer that gets assigned the number of constraints returned by the function (may be <code>NULL</code>).
numiis	Sequence number of the IIS or value 0 to access the IIS approximation.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following prints out the variable and constraint names of the first IIS found for an infeasible LP problem.

```
XPRBprob expl2;
XPRBctr *iisctr;
XPRBvar *iisvar;
int numv, numc;
expl2 = XPRBnewprob("example2");
...
XPRBsolve(expl2, "");
if(XPRBgetlpstat(expl2)==XPRB_LP_INFEAS)
{
    XPRBgetiis(expl2, &iisvar, &numv, &iisctr, &numc, 1);
    printf("Variables: ");          /* Print all variables */
    for(i=0;i<numv;i++) printf("%s ", XPRBgetvarname(iisvar[i]));
    printf("\n");
    free(iisvar);                  /* Free the array of variables */
    printf("Constraints: ");        /* Print all constraints */
    for(i=0;i<numc;i++) printf("%s ", XPRBgetctrname(iisctr[i]));
    printf("\n");
    free(iisctr);                  /* Free the array of constraints */
}
```

Further information

1. This function returns the variables and constraints forming an IIS (irreducible infeasible set) in an infeasible LP problem. The number of independent IIS identified by Xpress-Optimizer can be obtained with function `XPRBgetnumiis`.
2. The arrays of variables and constraints that are allocated by this function must be freed by the user's program.
3. The counting of IIS starts at 1. Value 0 for the argument `numiis` returns the information about the IIS approximation. Negative values or values larger than the number of IIS identified for the problem return 0 for the numbers of variables and constraints.

Related topics

`XPRBgetnumiis`, `XPRBgetlpstat`.

XPRBgetlim

Purpose

Get the integer limit for a partial integer or the semi-continuous limit for a semi-continuous or semi-continuous integer variable.

Synopsis

```
int XPRBgetlim(XPRBvar var, double *lim);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>lim</code>	Limit value.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBvar x3;  
double vlim;  
...  
x3 = XPRBnewvar(prob, XPRB_SC, "abc4", 0, 50);  
XPRBgetlim(x3, &vlim);
```

This obtains the lower bound of the continuous part of the variable `x3`.

Further information

This function returns the currently defined integer limit for a partial integer variable or the lower semi-continuous limit for a semi-continuous or semi-continuous integer variable.

Related topics

[XPRBfixvar](#), [XPRBgetbounds](#), [XPRBsetlb](#), [XPRBsetlim](#), [XPRBsetub](#).

XPRBgetlpstat

Purpose

Get the LP status.

Synopsis

```
int XPRBgetlpstat(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0	the problem has not been loaded, or error;
XPRB_LP_OPTIMAL	LP optimal;
XPRB_LP_INFEAS	LP infeasible;
XPRB_LP_CUTOFF	the objective value is worse than the cutoff;
XPRB_LP_UNFINISHED	LP unfinished;
XPRB_LP_UNBOUNDED	LP unbounded;
XPRB_LP_CUTOFF_IN_DUAL	LP cutoff in dual.
XPRB_LP_UNSOLVED	QP problem matrix is not semi-definite.

Example

The following returns the current LP status.

```
XPRBprob expl2;  
int status;  
  
...  
expl2 = XPRBnewprob("example2");  
XPRBsolve(expl2, "1");  
status = XPRBgetlpstat(expl2);
```

Further information

The return value of this function provides LP status information from the Xpress-Optimizer.

Related topics

[XPRBgetmipstat](#), [XPRBgetprobat](#).

XPRBgetmipstat

Purpose

Get the MIP status.

Synopsis

```
int XPRBgetmipstat(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

<code>XPRB_MIP_NOT_LOADED</code>	problem has not been loaded, or error;
<code>XPRB_MIP_LP_NOT_OPTIMAL</code>	LP has not been optimized;
<code>XPRB_MIP_LP_OPTIMAL</code>	LP has been optimized;
<code>XPRB_MIP_NO_SOL_FOUND</code>	global search incomplete — no integer solution found;
<code>XPRB_MIP_SOLUTION</code>	global search incomplete, although an integer solution has been found;
<code>XPRB_MIP_INFEAS</code>	global search complete, but no integer solution found;
<code>XPRB_MIP_OPTIMAL</code>	global search complete and an integer solution has been found.

Example

The following returns the current MIP status.

```
XPRBprob expl2;  
int status;  
expl2 = XPRBnewprob("example2");  
...  
XPRBsolve(expl2, "g");  
status = XPRBgetmipstat(expl2);
```

Further information

This function returns the global (MIP) status information from the Xpress-Optimizer.

Related topics

[XPRBgetlpstat](#), [XPRBgetprobstat](#).

XPRBgetmodcut

Purpose

Get the type of a constraint.

Synopsis

```
int XPRBgetmodcut(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

0	an ordinary constraint;
1	a model cut;
-1	an error has occurred.

Example

```
XPRBprob prob;
XPRBctr ctrl;
int mstat;

...
ctrl = XPRBnewctr(prob, "r1", XPRB_E);
mstat = XPRBgetmodcut(ctrl);
```

This determines whether `ctrl` is an ordinary constraint or a model cut.

Further information

This function indicates whether the given constraint is a model cut or an ordinary constraint.

Related topics

[XPRBsetmodcut](#).

XPRBgetnumiis

Purpose

Get the number of independent IIS in an infeasible LP problem.

Synopsis

```
int XPRBgetnumiis(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

Number of independent IIS found by Xpress-Optimizer, or a negative value in case of error.

Example

The following gets the number of IIS for a problem.

```
XPRBprob expl2;  
int num;  
expl2 = XPRBnewprob("example2");  
...  
XPRBsolve(expl2, "");  
if(XPRBgetlpstat(expl2)==XPRB_LP_INFEAS)  
    num = XPRBgetnumiis(expl2);
```

Further information

This function returns the number of independent IIS (irreducible infeasible sets) of an infeasible LP problem. After retrieving the number of IIS, the variables and constraints in each set can be obtained with function [XPRBgetiis](#).

Related topics

[XPRBgetiis](#), [XPRBgetlpstat](#).

XPRBgetobjval

Purpose

Get the objective function value.

Synopsis

```
double XPRBgetobjval(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

Current objective function value, default and error return value: 0.

Example

The following provides an example of retrieving the objective function value.

```
XPRBprob expl2;  
double objval;  
expl2 = XPRBnewprob("example2");  
...  
XPRBsolve(expl2, "l");  
objval = XPRBgetobjval(expl2);
```

Further information

This function returns the current objective function value from the Xpress-Optimizer. If it is called after completion of a global search and an integer solution has been found (that is, if function `XPRBgetmipstat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value of the best integer solution. In all other cases, including during a global search, it returns the solution value of the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBsync` with the flag `XPRB_XPRS_SOL`.

Related topics

`XPRBgetdual`, `XPRBgetrcost`, `XPRBgetsol`, `XPRBgetslack`, `XPRBgetact`, `XPRBsync`.

XPRBgetprobname

Purpose

Get the name of the specified problem.

Synopsis

```
const char *XPRBgetprobname(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

Name of the problem if function executed successfully, `NULL` otherwise.

Example

```
XPRBprob expl2;  
const char *pbname;  
expl2 = XPRBnewprob("example2");  
pbname = XPRBgetprobname(expl2);  
printf("%s", pbname);
```

This returns the name of the active problem and prints as output, `example2`.

Related topics

[XPRBdelprob](#), [XPRBnewname](#), [XPRBnewprob](#).

XPRBgetprobat

Purpose

Get the problem status.

Synopsis

```
int XPRBgetprobat(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

Bit-encoded BCL status information:

<code>XPRB_GEN</code>	the matrix has been generated;
<code>XPRB_DIR</code>	directives have been added;
<code>XPRB_MOD</code>	the problem has been modified;
<code>XPRB_SOL</code>	the problem has been solved.

Example

The following retrieves the current problem status and (re)solves the problem if it has been modified.

```
XPRBprob expl2;  
int status;  
...  
expl2 = XPRBnewprob("example2");  
status = XPRBgetprobat(expl2);  
if((status&XPRB_MOD)==XPRB_MOD)  
    XPRBsolve(expl2, "");
```

Further information

This function returns the current BCL problem status. Note that the problem status uses bit-encoding contrary to the LP and MIP status information, because several states may apply at the same time.

Related topics

[XPRBgetlpstat](#), [XPRBgetmipstat](#).

XPRBgetrange

Purpose

Get the range values for a range constraint.

Synopsis

```
int XPRBgetrange(XPRBctr ctr, double *bdl, double *bdu);
```

Arguments

<code>ctr</code>	Reference to a range constraint.
<code>bdl</code>	Lower bound on the range constraint. May be <code>NULL</code> if not required.
<code>bdu</code>	Upper bound on the range constraint. May be <code>NULL</code> if not required.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBctr ctr2;  
XPRBarrvar ty1;  
double bdl, bdu;  
  
...  
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);  
ctr2 = XPRBnewsum(prob, "r2", ty1, XPRB_E, 9);  
XPRBgetrange(ctr2, &bdl, &bdu);
```

This obtains the range values for `ctr2`.

Further information

This function returns the range values of the given constraint. If `bdl` or `bdu` is set to `NULL`, no value is returned into the corresponding argument.

Related topics

[XPRBsetrange](#).

XPRBgetrcost

Purpose

Get reduced cost value for a variable.

Synopsis

```
double XPRBgetrcost(XPRBvar var);
```

Argument

`var` Reference to a variable.

Return value

Reduced cost value for the variable, 0 in case of an error.

Example

```
XPRBprob expl2;  
XPRBvar x1;  
double rcval;  
...  
expl2 = XPRBnewprob("example2");  
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 1, 100);  
XPRBsolve(expl2, "1");  
rcval = XPRBgetrcost(x1);
```

This retrieves the reduced cost value for the variable `x1` in the solution to the LP problem.

Further information

This function returns the reduced cost value for a variable. The user may wish to test first whether this variable is part of the problem, for instance by checking that the column number is non-negative.

If this function is called after completion of a global search and an integer solution has been found (that is, if function [XPRBgetmipstat](#) returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the reduced cost value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to [XPRBsync](#) with the flag `XPRB_XPRS_SOL`.

Related topics

[XPRBgetdual](#), [XPRBgetobjval](#), [XPRBgetslack](#), [XPRBgetsol](#), [XPRBsync](#).

XPRBgetrhs

Purpose

Get the right hand side value of a constraint.

Synopsis

```
double XPRBgetrhs(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

Right hand side value of the constraint, 0 in case of an error.

Example

The following retrieves the right hand side value of the constraint `ctr1`.

```
XPRBprob prob;  
XPRBctr ctr1;  
double rhs;  
  
...  
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);  
rhs = XPRBgetrhs(ctr1);
```

Further information

This function returns the right hand side value (*i.e.* the constant term) of a previously defined constraint. The default right hand side value is 0. If the given constraint is a ranged constraint this function returns its upper bound.

Related topics

[XPRBaddterm](#), [XPRBgetctrtype](#), [XPRBsetctrtype](#), [XPRBsetterm](#).

XPRBgetrownum

Purpose

Get the row number for a constraint.

Synopsis

```
int XPRBgetrownum(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

Row number (non-negative value), or a negative value.

Example

The following gets the row number of `ctr1`.

```
XPRBprob prob;
XPRBctr ctr1;
...
int rindex;
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
rindex = XPRBgetrownum(ctr1);
```

Further information

This function returns the matrix row number of a constraint. If the matrix has not yet been generated or the constraint is not part of the matrix (constraint type `XPRB_N` or no non-zero terms) then the return value is negative. To check whether the matrix has been generated, use function `XPRBgetprobstat`. The counting of row numbers starts with 0.

Related topics

`XPRBdelctr`, `XPRBnewctr`.

XPRBgetsense

Purpose

Get the sense of the objective function.

Synopsis

```
int XPRBgetsense(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

<code>XPRB_MAXIM</code>	the objective function is to be maximized;
<code>XPRB_MINIM</code>	the objective function is to be minimized;
<code>-1</code>	an error has occurred.

Example

The following returns the sense of the problem `expl2`.

```
XPRBprob expl2;  
int dir;  
...  
expl2 = XPRBnewprob("example2");  
dir = XPRBgetsense(expl2);
```

Further information

This function returns the objective sense (maximization or minimization). The sense is set to minimization by default and may be changed with functions [XPRBsetsense](#), [XPRBminim](#), and [XPRBmaxim](#).

Related topics

[XPRBmaxim](#), [XPRBminim](#), [XPRBsetsense](#), [XPRBsolve](#).

XPRBgetslack

Purpose

Get slack value for a constraint.

Synopsis

```
double XPRBgetslack(XPRBctr ctr);
```

Argument

`ctr` Reference to a constraint.

Return value

Slack value for the constraint, 0 in case of an error.

Example

```
XPRBprob expl2;
XPRBctr ctr2;
XPRBarrvar ty1;
double slack;
...
expl2 = XPRBnewprob("example2");
ty1 = XPRBnewarrvar(expl2, 5, XPRB_PL, "arry1", 0, 500);
ctr2 = XPRBnewsum(expl2, "r2", ty1, XPRB_E, 9);
XPRBsolve(expl2, "1");
slack = XPRBgetslack(ctr2);
```

This obtains the slack value for the constraint `ctr2`.

Further information

This function returns the slack value for a constraint. The user may wish to test first whether this constraint is part of the problem, for instance by checking that the row number is non-negative.

If this function is called after completion of a global search and an integer solution has been found (that is, if function [XPRBgetmipstat](#) returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the slack value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to [XPRBsync](#) with the flag `XPRB_XPRS_SOL`.

Related topics

[XPRBgetact](#), [XPRBgetdual](#), [XPRBgetobjval](#), [XPRBgetrcost](#), [XPRBgetsol](#), [XPRBsync](#).

XPRBgetsol

Purpose

Get solution value for a variable.

Synopsis

```
double XPRBgetsol(XPRBvar var);
```

Argument

`var` Reference to a variable.

Return value

Primal solution value for the variable, 0 in case of an error.

Example

```
XPRBprob expl2;
XPRBvar x1;
double solval;
...
expl2 = XPRBnewprob("example2");
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 1, 100);
XPRBsolve(expl2, "1");
solval = XPRBgetsol(x1);
```

The example retrieves the LP solution value for the variable `x1`.

Further information

1. This function returns the current solution value for a variable. The user may wish to test first whether this variable is part of the problem, for instance by checking that the column number is non-negative.
If this function is called after completion of a global search and an integer solution has been found (that is, if function `XPRBgetmipstat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value of the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the solution value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBsync` with the flag `XPRB_XPRS_SOL`.
2. Note that “integer solution” means “solution within the integer feasibility limits”, that means for any comparison of solution values the current Optimizer tolerance settings have to be taken into account. So care must be taken when handling the solution values of integer variables. For example, you cannot simply treat the value as an integer, because a value such as 0.999998, may well be truncated to zero. Instead, you must make sure you round the value to the nearest integer.

Related topics

`XPRBgetact`, `XPRBgetdual`, `XPRBgetobjval`, `XPRBgetrcost`, `XPRBgetslack`, `XPRBsync`.

XPRBgetsosname

Purpose

Get the name of a SOS.

Synopsis

```
const char *XPRBgetsosname(XPRBsos sos);
```

Argument

`sos` Reference to a previously created SOS.

Return value

Name of the SOS if function executed successfully, `NULL` otherwise.

Example

```
XPRBprob prob;  
XPRBsos set1;  
...  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
printf("%s\n", XPRBgetsosname(set1));
```

The prints "sos1" as output.

Further information

This function returns the name of a SOS. If the user has not defined a name the default name generated by BCL is returned.

Related topics

[XPRBdelsos](#), [XPRBgetsostype](#), [XPRBnewsos](#).

XPRBgetsostype

Purpose

Get the type of a SOS.

Synopsis

```
int XPRBgetsostype(XPRBsos sos);
```

Argument

`sos` Reference to a previously created SOS.

Return value

`XPRB_S1` a Special Ordered Set of type 1;

`XPRB_S2` a Special Ordered Set of type 2;

`-1` an error has occurred.

Example

```
XPRBprob prob;  
XPRBsos set1;  
char stype;  
...  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
stype = XPRBgetsostype(set1);
```

This returns the type of the SOS `set1`.

Further information

The function returns the type of a SOS.

Related topics

[XPRBdelsos](#), [XPRBgetsosname](#), [XPRBnewsos](#).

XPRBgettime

Purpose

Get the running time.

Synopsis

```
int XPRBgettime(void);
```

Return value

System time measure in milliseconds.

Example

The following provides an example of obtaining the running time for code.

```
int starttime;
starttime = XPRBgettime();
...
printf("Time: %g sec", (XPRBgettime()-starttime)/1000);
```

Further information

This function returns the system time measure in milliseconds. The absolute value is system-dependent. To measure the execution time of a program, this function can be used to calculate the difference between the start time and the time at the desired point in the program.

Related topics

[XPRBgetversion](#).

XPRBgetvarlink

Purpose

Get the interface pointer of a variable.

Synopsis

```
void *XPRBgetvarlink(XPRBvar var);
```

Argument

`var` Reference to a BCL variable

Return value

Pointer to an interface object, or `NULL`.

Example

Set the interface pointer of variable `x1` to `vlink`:

```
XPRBprob prob;  
XPRBvar x1;  
void *vlink;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
vlink = XPRBsetvarlink(x1);
```

Further information

This function returns the interface pointer of a variable to the indicated object. It may be used to establish a connection between a variable in BCL and some other external program.

Related topics

[XPRBsetvarlink](#), [XPRBdefcbdelvar](#).

XPRBgetvarname

Purpose

Get the name of a variable.

Synopsis

```
const char *XPRBgetvarname(XPRBvar var);
```

Argument

`var` BCL reference to a variable.

Return value

Name of the variable if function executed successfully, `NULL` otherwise.

Example

This example prints the retrieved variable name.

```
XPRBprob prob;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
printf("%s\n", XPRBgetvarname(x1));
```

Further information

This function returns the name of a variable. If the user has not defined a name the default name generated by BCL is returned.

Related topics

[XPRBgetarrvarname](#), [XPRBgetvartype](#), [XPRBnewvar](#), [XPRBsetvartype](#).

XPRBgetvarrng

Purpose

Get ranging information for a variable.

Synopsis

```
double XPRBgetvarrng(XPRBvar var, int rngtype);
```

Arguments

var	Reference to variable.
rngtype	The type of ranging information sought. This is one of: XPRB_UPACT upper activity; XPRB_LOACT lower activity; XPRB_UUP upper unit cost; XPRB_UDN lower unit cost; XPRB_UCOST upper cost; XPRB_LCOST lower cost.

Return value

Ranging information of the required type.

Example

This example retrieves the upper cost value for a variable.

```
XPRBprob expl2;  
XPRBvar x1;  
double ucval;  
expl2 = XPRBnewprob("example2");  
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 1, 100);  
...  
XPRBsolve("expl2, 1");  
ucval = XPRBgetvarrng(x1, XPRB_UCOST);
```

Further information

This method can only be used after solving an LP problem. Ranging information for MIP problems can be obtained by fixing all discrete variables to their solution values and re-solving the resulting LP problem.

Related topics

[XPRBnewvar](#), [XPRBgetctrng](#).

XPRBgetvartype

Purpose

Get the type of a variable.

Synopsis

```
int XPRBgetvartype(XPRBvar var);
```

Argument

`var` BCL reference to a variable.

Return value

<code>XPRB_PL</code>	continuous;
<code>XPRB_BV</code>	binary;
<code>XPRB_UI</code>	general integer;
<code>XPRB_PI</code>	partial integer;
<code>XPRB_SC</code>	semi-continuous;
<code>XPRB_SI</code>	semi-continuous integer;
<code>-1</code>	an error has occurred.

Example

```
XPRBprob prob;  
XPRBvar x1;  
char vtype;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
vtype = XPRBgetvartype(x1);
```

This returns the type of variable `x1`.

Further information

If the function exits successfully, the variable type is returned.

Related topics

[XPRBnewvar](#), [XPRBsetvartype](#).

XPRBgetversion

Purpose

Get the version number of BCL.

Synopsis

```
const char *XPRBgetversion(void);
```

Return value

BCL version number if function executed successfully, `NULL` otherwise.

Example

The following obtains the BCL version number, displaying output similar to 1.1.0.

```
const char *version;
version = XPRBgetversion();
printf("%s", version);
```

Further information

This function returns the version number of BCL. This information is required if the user is reporting a problem.

Related topics

[XPRBgettime](#).

XPRBgetXPRSProb

Purpose

Returns an `XPRSProb` problem reference for a problem defined in BCL and subsequently loaded into the Xpress-Optimizer.

Synopsis

```
XPRSProb XPRBgetXPRSProb(XPRBprob prob);
```

Argument

`prob` The current BCL problem.

Return value

Reference to a problem in Xpress-Optimizer if function executed successfully, `NULL` otherwise.

Example

The Xpress-Optimizer problem reference needs to be retrieved to access control parameters and optimizer problem attributes:

```
XPRBprob bcl_prob;  
XPRSProb opt_prob;  
  
bcl_prob = XPRBnewprob("MyProb");  
...  
XPRBloadmat(bcl_prob);  
opt_prob = XPRBgetXPRSProb(bcl_prob);  
XPRSsetintcontrol(opt_prob, XPRS_PRESOLVE, 0);
```

Further information

The optimizer problem returned by this function may be different from the one loaded in BCL if the solution algorithms have not been called (and the problem has not been loaded explicitly) after the last modifications to the problem in BCL, or if any modifications have been carried out directly on the problem in the optimizer.

Related topics

[XPRBloadmat](#), [XPRBnewprob](#), [Chapter B](#).

XPRBinit

Purpose

Initialize BCL.

Synopsis

```
int XPRBinit(void);
```

Return value

0	function executed successfully,
1	an error has occurred,
32	BCL has been set running in Student mode.

Example

```
XPRBseterrctrl(0);
if(XPRBinit())
    printf("BCL has not been initialized correctly.
           Please check your Xpress-MP licenses.");
```

This switches to user error handling and initializes BCL (or performs license test).

Further information

1. This function explicitly initializes BCL, that is it tests whether a license for running this software is available. It is possible to run BCL with a student license; this mode implies restrictions to the available functionality and to the accepted problem size.
2. The initialization is also performed by function [XPRBnewprob](#) so that usually there is no need to call this explicit initialization. This function may be used if the embedding of BCL into some larger application requires a test of the license at an earlier stage, before even creating any model. Note that this function also initializes Xpress-Optimizer, so that it is usually not necessary to call [XPRSinit](#) separately (the latter is only required if one wishes to continue using the optimizer after terminating BCL).

Related topics

[XPRBfree](#), [XPRBnewprob](#), [XPRSinit](#) (see Optimizer Reference Manual).

XPRBloadbasis

Purpose

Load a previously saved basis.

Synopsis

```
int XPRBloadbasis(XPRBbasis basis);
```

Argument

`basis` Reference to a previously saved basis.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code saves the current basis prior to some matrix changes, before subsequently reloading the saved basis to solve the linear relaxation.

```
XPRBprob expl2;
XPRBbasis basis;

...
expl2 = XPRBnewprob("example2");
XPRBsolve(expl2, "l");
basis = XPRBsavebasis(expl2);

...
XPRBloadmat(expl2);
XPRBloadbasis(basis);
XPRBdelbasis(basis);
XPRBsolve(expl2, "l");
```

Further information

This function loads a basis for the current problem. The basis must have been saved using function [XPRBsavebasis](#). It is *not* possible to load a basis saved for any other problem than the current one, even if the problems are similar. This function takes into account that the problem may have been modified (addition/deletion of variables and constraints) since the basis has been stored. For reading a basis from a file, the Optimizer library function [XPRSreadbasis](#) may be used. Note that the problem has to be loaded explicitly (function [XPRBloadmat](#)) before the basis is re-input with [XPRBloadbasis](#). Furthermore, if the reference to a basis is not used any more it should be deleted using function [XPRBdelbasis](#).

Related topics

[XPRBdelbasis](#), [XPRBsavebasis](#), [XPRSreadbasis](#) (see Optimizer Reference Manual), [XPRS-writebasis](#) (see Optimizer Reference Manual).

XPRBloadmat

Purpose

Load the problem into the Xpress-Optimizer.

Synopsis

```
int XPRBloadmat(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

Here the matrix is generated for problem `expl2`.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBloadmat(expl2);
```

Further information

This function calls the Optimizer library functions `XPRSloadlp`, `XPRSloadqp`, `XPRSloadglobal`, or `XPRSloadqglobal` to transform the current BCL problem definition into a matrix in the Xpress-Optimizer. Empty rows and columns are deleted before generating the matrix. Semi-continuous (integer) variables are preprocessed: if a lower bound value greater than 0 is given, then the variable is treated like a continuous (resp. integer) variable. Variables that belong to the problem but do not appear in the matrix receive negative column numbers. Usually, it is *not* necessary to call this function explicitly because BCL automatically does this conversion whenever it is required. To force matrix reloading, a call to this function needs to be preceded by a call to `XPRBsync` with the flag `XPRB_XPRS_PROB`.

Related topics

`XPRBsync`, `XPRBgetXPRSpb`, Appendix B.

XPRBmaxim

Purpose

Maximize the objective function for the given problem.

Synopsis

```
int XPRBmaxim(XPRBprob prob, const char *flags);
```

Arguments

prob	Reference to a problem.
flags	Choice of the solution algorithm, which may be one of: <ul style="list-style-type: none">" " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);"d" solve the problem using the dual simplex algorithm;"p" solve the problem using the primal simplex algorithm;"b" solve the problem using the Newton barrier algorithm;"n" use the network solver (LP only);"l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);"g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value

0 if function executed successfully, 1 otherwise.

Example

Maximize the LP problem using a Newton-Barrier algorithm.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBmaxim(expl2, "b");
```

Further information

This function selects and starts the Xpress-Optimizer solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, e.g. "dg. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem it is regenerated automatically prior to the start of the algorithm. Matrix reloading can also be forced by calling [XPRBsync](#) before the optimization. Before solving a problem, the objective function must be selected with [XPRBsetobj](#). Note that if you use an incomplete global search you should finish your program with a call to the Optimizer library function [XPRSinitglobal](#) in order to remove all search tree information that has been stored. Otherwise you may not be able to rerun your program.

Related topics

[XPRBgetobjval](#), [XPRBgetsol](#), [XPRBminim](#), [XPRBsetsense](#), [XPRBsm Maxim](#) (see Optimizer Reference Manual).

XPRBminim

Purpose

Minimize the objective function for the given problem.

Synopsis

```
int XPRBminim(XPRBprob prob, char *flags);
```

Arguments

prob	Reference to a problem.
flags	Choice of the solution algorithm, which may be one of: <ul style="list-style-type: none">" " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);"d" solve the problem using the dual simplex algorithm;"p" solve the problem using the primal simplex algorithm;"b" solve the problem using the Newton barrier algorithm;"n" use the network solver (LP only);"l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);"g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code minimizes the objective function of `expl2` using the Newton barrier algorithm.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBminim(expl2, "b");
```

Further information

This function selects and starts the Xpress-Optimizer solution algorithm. The flags indicating the algorithm choice may be combined where it makes sense, e.g. "dg". If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem it is regenerated automatically prior to the start of the algorithm. Matrix reloading can also be forced by calling `XPRBsync` before the optimization. Before solving a problem, the objective function must be selected with `XPRBsetobj`. Note that if you use an incomplete global search you should finish your program with a call to the Optimizer library function `XPRSinitglobal` in order to remove search tree information that has been stored, or else you may not be able to rerun your program.

Related topics

`XPRBgetobjval`, `XPRBgetsol`, `XPRBmaxim`, `XPRBsetsense`, `XPRBsolve`, `XPRBsync`, `XPRBminim` (see Optimizer Reference Manual).

XPRBnewarrsum

Purpose

Create a sum constraint with individual coefficients.

Synopsis

```
XPRBctr XPRBnewarrsum(XPRBprob prob, const char *name,  
                      XPRBarrvar av, double *cof, int qrtype, double rhs);
```

Arguments

prob	Reference to a problem.
name	The constraint name (of unlimited length). May be <code>NULL</code> if not required.
av	Reference to an array of variables.
cof	Array of constant coefficients for all elements of <code>av</code> . It must be at least the same size as <code>av</code> .
qrtype	Type of the constraint, which must be one of: XPRB_L 'less than or equal to' constraint; XPRB_G 'greater than or equal to' constraint; XPRB_E equality constraint; XPRB_N non-binding constraint (objective function).
rhs	The right hand side value of the constraint.

Return value

Reference to the new constraint if function executed successfully, `NULL` otherwise.

Example

The following creates the constraint $\sum_{i=0}^4 c_i \cdot ty1_i \geq 7.0$.

```
XPRBprob prob;  
XPRBctr ctr4;  
XPRBarrvar ty1;  
double c[] = {2.5, 4.0, 7.2, 3.0, 1.8};  
...  
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);  
ctr4 = XPRBnewarrsum(prob, "r4", ty1, c, XPRB_G, 7.0);
```

Further information

This function creates a linear constraint consisting of the sum over variables multiplied by the coefficients indicated by array `cof`. This function replaces [XPRBnewctr](#) and [XPRBaddterm](#). If the indicated name is already in use, BCL adds an index to it. If no constraint name is given, BCL generates a default name starting with `CTR`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)

Related topics

[XPRBdelctr](#), [XPRBnewctr](#), [XPRBnewprec](#), [XPRBnewsum](#), [XPRBsetdictionarysize](#).

XPRBnewarrvar

Purpose

Create a one-dimensional array of variables.

Synopsis

```
XPRBarrvar XPRBnewarrvar(XPRBprob prob, int nbvar, int type,
                        const char *name, double bdl, double bdu);
```

Arguments

prob	Reference to a problem.
nbvar	Size of the array of variables.
type	Type of the variables, which may be one of: XPRB_PL continuous; XPRB_BV binary; XPRB_UI general integer; XPRB_PI partial integer; XPRB_SC semi-continuous; XPRB_SI semi-continuous integer.
name	The array name. May be <code>NULL</code> if not required.
bdl	Variable lower bound.
bdu	Variable upper bound.

Return value

Reference to the new array of variables if function executed successfully, `NULL` otherwise.

Example

The following defines an array of ten continuous variables between 0 and 500, with names beginning `arry1` followed by a counter.

```
XPRBprob prob;
XPRBarrvar ty1;
...
ty1 = XPRBnewarrvar(prob, 10, XPRB_PL, "arry1", 0, 500);
```

Further information

1. This function creates a single-indexed array of variables. Individual bounds on variables may be changed afterwards using [XPRBsetlb](#) and [XPRBsetub](#), and variable types by using [XPRBsetvartype](#). The function returns the BCL reference to the array of variables. If `name` is defined, BCL generates names for the variables in the array by adding an index to the string. If no array name is given, BCL generates a default name starting with `av`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)
2. Either of the bounds `XPRB_INFINITY` or `-XPRB_INFINITY` for plus or minus infinity may be used for the arguments `bdu` and `bdl`.

Related topics

[XPRBdelarrvar](#), [XPRBendarrvar](#), [XPRBstartarrvar](#), [XPRBsetdictionarysize](#).

XPRBnewctr

Purpose

Create a new constraint.

Synopsis

```
XPRBctr XPRBnewctr(XPRBprob prob, const char *name, int qrtype);
```

Arguments

prob	Reference to a problem.
name	The constraint name (of unlimited length). May be <code>NULL</code> if not required.
type	Type of the constraint, which must be one of <code>XPRB_L</code> 'less than or equal to' inequality; <code>XPRB_G</code> 'greater than or equal to' inequality; <code>XPRB_E</code> equality; <code>XPRB_N</code> a non-binding row (objective function).

Return value

Reference to the new constraint if function executed successfully, `NULL` otherwise.

Example

The following creates a new equality constraint.

```
XPRBprob prob;  
XPRBctr ctrl;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);
```

Further information

This function creates a new constraint and returns the reference to this constraint, *i.e.*, the constraint's model name. It has to be called before `XPRBaddterm` or `XPRBaddqterm` is used to add terms to the constraint. Range constraints can first be created with any type and then converted using the function `XPRBsetrange`. If the indicated name is already in use, BCL adds an index to it. If no constraint name is given, BCL generates a default name starting with `CTR`. (The generation of unique names will only take place if the names dictionary is enabled, see `XPRBsetdictionarysize`.)

Related topics

`XPRBaddterm`, `XPRBdelctr`, `XPRBdelterm`, `XPRBsetdictionarysize`.

XPRBnewcut

Purpose

Create a new cut.

Synopsis

```
XPRBcut XPRBnewcut(XPRBprob prob, int qrtype, int mtype);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>qrtype</code>	Type of the cut: <code>XPRB_L</code> \leq (inequality) <code>XPRB_G</code> \geq (inequality) <code>XPRB_E</code> = (equation)
<code>mtype</code>	Cut classification or identification number.

Return value

Reference to the new cut of type `xbcut` if function executed successfully, `NULL` otherwise.

Example

The example shows how to create a new equality cut.

```
XPRBcut cut1;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
```

Further information

This function creates a new cut and returns the reference to this cut, *i.e.* the cut's model name. It has to be called before `XPRBaddcutterm` is used to add terms to the cut.

Related topics

`XPRBaddcutterm`, `XPRBdelcut`, `XPRBaddcuts`.

XPRBnewcutarrsum

Purpose

Create a sum cut with individual coefficients ($\sum_i c_i \cdot x_i$).

Synopsis

```
XPRBcut XPRBnewcutarrsum(XPRBprob prob, XPRBarrvar av, double *cof, char
qrtype, double rhs, int mtype);
```

Arguments

prob	Reference to a problem.
av	Reference to an array of variables.
cof	Array of constant coefficients for all elements of (at least size of <code>av</code>).
qrtype	Type of the cut: XPRB_L \leq (inequality) XPRB_G \geq (inequality) XPRB_E $=$ (equation)
rhs	RHS value of the cut.
mtype	Cut classification or identification number.

Return value

Reference to the new cut if function executed successfully, `NULL` otherwise.

Example

The following creates the inequality constraint $\sum_{i=0}^4 c_i \cdot ty1_i \geq 7$.

```
XPRBcut cut4;
XPRBarrvar ty1;
double c[] = {2.5, 4.0, 7.2, 3.0, 1.8};
ty1 = XPRBnewarrvar(5, XPRB_PL, "array1", 0, 500);
cut4 = XPRBnewcutarrsum(ty1, c, XPRB_G, 7.0, 18);
```

Further information

This function creates a cut consisting of the sum over variables multiplied by the coefficients indicated by array `cof`. This function replaces `XPRBnewcut` and `XPRBaddcutterm`.

Related topics

`XPRBnewcut`, `XPRBaddcutterm`.

XPRBnewcutprec

Purpose

Create a precedence cut ($v_1 + dur \leq v_2$).

Synopsis

```
XPRBcut XPRBnewcutprec(XPRBprob prob, XPRBvar v1, double dur, XPRBvar v2,  
    int mtype);
```

Arguments

prob	Reference to a problem.
v1, v2	References to two variables.
dur	Double or integer constant.
mtype	Cut classification or identification number.

Return value

Reference to the newly created cut if function executed successfully, `NULL` otherwise.

Example

The following creates the inequality constraint $ty1_2 + 5.4 \leq ty1_4$.

```
XPRBcut cut5;  
XPRBarrvar ty1;  
ty1 = XPRBnewarrvar(5, XPRB_PL, "arry1", 0, 500);  
cut5 = XPRBnewcutprec(ty1[2], 5.4, ty1[4], 5);
```

Further information

This function creates a so-called precedence constraint (where the variable plus constant is not larger than a second variable). This function replaces `XPRBnewcut` and `XPRBaddcutterm`.

Related topics

`XPRBnewcut`, `XPRBaddcutterm`.

XPRBnewcutsum

Purpose

Create a sum cut ($\sum_i x_i$).

Synopsis

```
XPRBcut XPRBnewcutsum(XPRBprob prob, XPRBarrvar tv, char qrtype, double
                      rhs, int mtype);
```

Arguments

prob	Reference to a problem.
av	Reference to an array of variables.
qrtype	Type of the cut: XPRB_L \leq (inequality) XPRB_G \geq (inequality) XPRB_E $=$ (equation)
rhs	RHS value of the cut.
mtype	Cut classification or identification number.

Return value

Reference to the new cut if function executed successfully, `NULL` otherwise.

Example

Create the equality constraint $\sum_{i=0}^4 ty1_i = 9$.

```
XPRBcut cut2;
XPRBarrvar ty1;
ty1 = XPRBnewarrvar(5, XPRB_PL, "arry1", 0, 500);
cut2 = XPRBnewcutsum(ty1, XPRB_E, 9, 3);
```

Further information

This function creates a simple sum constraint over all entries of an array of variables. It replaces calls to `XPRBnewcut` and `XPRBaddcutterm`.

Related topics

`XPRBnewcut`, `XPRBaddcutterm`.

XPRBnewidxset

Purpose

Create a new index set.

Synopsis

```
XPRBidxset XPRBnewidxset(XPRBprob prob, const char *name, int maxsize);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>name</code>	Name of the index set to be created. May be <code>NULL</code> if not required.
<code>maxsize</code>	Maximum size of the index set.

Return value

Reference to the new index set if function executed successfully, `NULL` otherwise.

Example

The following defines an index set with space for 100 entries.

```
XPRBprob prob;  
XPRBidxset iset;  
...  
iset = XPRBnewidxset(prob, "Set", 100);
```

Further information

This function creates a new index set. Note that the indicated size `maxsize` corresponds to the space allocated initially to the set, but it is increased dynamically if need be. If the indicated set name is already in use, BCL adds an index to it. If no name is given, BCL generates a default name starting with `IDX`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)

Related topics

[XPRBaddidxel](#), [XPRBgetidxel](#), [XPRBgetidxsetname](#), [XPRBgetidxsetsize](#),
[XPRBsetdictionarysize](#).

XPRBnewname

Purpose

Compose a name string.

Synopsis

```
const char *XPRBnewname(const char *format, ...);
```

Arguments

<code>format</code>	String indicating the printing format using standard C conventions (see the documentation of <code>printf</code> in a C manual for a complete list of format options). Simple formatting options are of the form <code>%n</code> where <code>n</code> may be, for instance, one of <code>c</code> single character; <code>d</code> integer; <code>g</code> double; <code>s</code> string of characters.
<code>...</code>	items composing the name string according to the format specification in the format string; separated by commas.

Return value

String of characters.

Example

This example finds the variable with name `xab15`.

```
XPRBprob prob;  
char a[] = "ab";  
int i = 15;  
XPRBvar x1;  
...  
x1 = XPRBgetbyname(prob, XPRBnewname("x%s%d", a, i), XPRB_VAR);
```

Further information

1. This function simplifies the composition of names for BCL objects. It is intended to be used as a parameter of other functions (wherever name strings are required). Unlike the standard C string functions, this function does not require any memory allocation by the user, and the string returned must not be freed by the user.
2. Names created with this function are limited to 128 characters. However, there is no restriction on the length of names for BCL objects in general.

Related topics

[XPRBdelprob](#), [XPRBgetprobname](#), [XPRBnewprob](#).

XPRBnewprec

Purpose

Create a precedence constraint $v1 + dur \leq v2$.

Synopsis

```
XPRBctr XPRBnewprec(XPRBprob prob, const char *name, XPRBvar v1,
                    double dur, XPRBvar v2);
```

Arguments

prob	Reference to a problem.
name	The constraint name (of unlimited length). May be <code>NULL</code> if not required.
v1	Reference to a variable.
dur	Double or integer constant.
v2	Reference to a variable.

Return value

Reference to the new constraint if function executed successfully, `NULL` otherwise.

Example

The following creates the inequality constraint $ty1_2 + 5.4 \leq ty1_4$.

```
XPRBprob prob;
XPRBctr ctr5;
XPRBarrvar ty1;
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
ctr5 = XPRBnewprec(prob, "r5", ty1[2], 5.4, ty1[4]);
```

Further information

This function creates a so-called precedence constraint (where the first variable plus constant is not larger than a second variable). This function replaces `XPRBnewctr` and `XPRBaddterm`. If the indicated name is already in use, BCL adds an index to it. If no constraint name is given, BCL generates a default name starting with `CTR`. (The generation of unique names will only take place if the names dictionary is enabled, see `XPRBsetdictionarysize`.)

Related topics

`XPRBnewarrsum`, `XPRBnewsum`, `XPRBsetdictionarysize`.

XPRBnewprob

Purpose

Initialize a new problem.

Synopsis

```
XPRBprob XPRBnewprob(const char *probname);
```

Argument

`probname` The problem name. May be `NULL` if not required.

Return value

Reference to a problem definition in BCL if function executed successfully, `NULL` otherwise.

Example

This example begins the definition of a new problem with the name `example2`.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");
```

Further information

1. This function needs to be called to create and initialize a new problem. This function initializes BCL and also Xpress-Optimizer; it is *not* necessary to call `XPRSinit` from the user's program. If the initialization does not find a valid license, BCL does not initialize. It is possible to run BCL with a student license; this mode implies restrictions to the available functionality and to the accepted problem size.
2. The name given to a problem determines the name and the location of the working files of Xpress-Optimizer. At the creation of a problem any existing working files of the same name are deleted. When solving several instances of a problem simultaneously the user must make sure to assign a different name to every instance. If no problem name is indicated, BCL creates a unique name including the full path to the temporary directory (Xpress-Optimizer creates its working files in the temporary directory).

Related topics

`XPRBdelprob`, `XPRBgetprobname`, `XPRBinit`.

XPRBnewsos

Purpose

Create a SOS.

Synopsis

```
XPRBsos XPRBnewsos(XPRBprob prob, const char *name, int type);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>name</code>	The name of the set.
<code>type</code>	The set type, which must be one of: <code>XPRB_S1</code> Special Ordered Set of type 1; <code>XPRB_S2</code> Special Ordered Set of type 2.

Return value

Reference to the new SOS if function executed successfully, `NULL` otherwise.

Example

The following creates an SOS of type 1, called `sos1`.

```
XPRBprob prob;  
XPRBsos set1;  
...  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);
```

Further information

This function creates a Special Ordered Set (SOS) of type 1 or 2 (abbreviated SOS1 and SOS2). It returns the address of the set that is taken as a parameter in the functions for adding set members, such as [XPRBaddsosel](#), deleting single elements [XPRBdelsosel](#) or the entire set [XPRBdelsos](#). If the indicated name is already in use, BCL adds an index to it. If no name is given for the set, BCL generates a default name starting with `SOS`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)

Related topics

[XPRBdelsos](#), [XPRBgetsosname](#), [XPRBgetsostype](#), [XPRBnewsosrc](#), [XPRBnewsosw](#), [XPRBsetdictionarysize](#).

XPRBnewsosrc

Purpose

Create a SOS, using a reference constraint.

Synopsis

```
XPRBsos XPRBnewsosrc(XPRBprob prob, const char *name, int type,
                     XPRBarrvar av, XPRBctr ctr);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>name</code>	Name of the set.
<code>type</code>	The set type, which must be one of: <code>XPRB_S1</code> Special Ordered Set of type 1; <code>XPRB_S2</code> Special Ordered Set of type 2.
<code>av</code>	Array of variables. May be <code>NULL</code> if not required.
<code>ctr</code>	Reference to a constraint which has been previously defined. May be <code>NULL</code> if not required.

Return value

Reference to the new SOS if function executed successfully, `NULL` otherwise.

Example

The following creates an SOS of type 2 with variables from the array `ty1`, and their coefficients in the constraint `ctr4`.

```
XPRBprob prob;
XPRBsos set2;
XPRBctr ctr4;
XPRBarrvar ty1;
double c[] = {2.5, 4.0, 7.2, 3.0, 1.8};
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
ctr4 = XPRBnewarrsum(prob, "r4", ty1, c, XPRB_G, 7.0);
set2 = XPRBnewsosrc(prob, "sos2", XPRB_S2, ty1, ctr4);
```

Further information

This function can be used instead of a stepwise SOS definition if the variables are available in the form of a single array and the model contains a constraint with nonzero coefficients for all variables which can serve as a reference constraint. If no reference constraint is indicated all weights are initialized to 1. If the indicated name is already in use, BCL adds an index to it. If no name is given for the set, BCL generates a default name starting with `SOS`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)

Related topics

[XPRBdelsos](#), [XPRBgetsosname](#), [XPRBgetsostype](#), [XPRBnewsos](#), [XPRBnewsosw](#), [XPRBsetdictionarysize](#).

XPRBnewsosw

Purpose

Create a SOS, using weights.

Synopsis

```
XPRBsos XPRBnewsosw(XPRBprob prob, const char *name, int type,
                    XPRBarrvar av, double *weight);
```

Arguments

prob	Reference to a problem.
name	The set name.
type	The set type, which must be one of: XPRB_S1 Special Ordered Set of type 1; XPRB_S2 Special Ordered Set of type 2.
av	An array of variables.
weight	An array of weights. May be <code>NULL</code> if not required.

Return value

Reference to the new SOS if function executed successfully, `NULL` otherwise.

Example

The following creates an SOS of type 1, with the variables in array `ty1` and weights, `cr`.

```
XPRBprob prob;
XPRBsos set1;
XPRBarrvar ty1;
double cr[] = {2.0, 13.0, 15.0, 6.0, 8.5};
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
set1 = XPRBnewsosw(prob, "sos1", XPRB_S1, ty1, cr);
```

Further information

This function can be used instead of a stepwise SOS definition using functions [XPRBnewsos](#) and [XPRBaddsosarrel](#), that is if the variables and their weights are available in the form of two arrays. If no weights are defined, the reference values of the variables are set to 1. If the indicated name is already in use, BCL adds an index to it. If no name is given for the set, BCL generates a default name starting with `SOS`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)

Related topics

[XPRBdelsos](#), [XPRBgetsosname](#), [XPRBnewsos](#), [XPRBnewsosrc](#), [XPRBsetdictionarysize](#).

XPRBnewsum

Purpose

Create a sum constraint.

Synopsis

```
XPRBctr XPRBnewsum(XPRBprob prob, const char *name, XPRBarrvar av,  
                   int type, double rhs);
```

Arguments

prob	Reference to a problem.
name	The constraint name (of unlimited length). May be <code>NULL</code> if not required.
av	Reference to an array of variables.
type	Type of the constraint, which must be one of: XPRB_L 'less than or equal to' constraint; XPRB_G 'greater than or equal to' constraint; XPRB_E equality; XPRB_N a non-binding row (objective function).
rhs	Right hand side value of the constraint.

Return value

Reference to the new constraint if function executed successfully, `NULL` otherwise.

Example

The following creates a new constraint, `ctr2`, given by $\sum_{i=0}^4 ty1_i = 9$.

```
XPRBprob prob;  
XPRBctr ctr2;  
XPRBarrvar ty1;  
...  
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);  
ctr2 = XPRBnewsum(prob, "r2", ty1, XPRB_E, 9);
```

Further information

This function creates a simple sum constraint over all entries of an array of variables. It replaces calls to [XPRBnewctr](#) and [XPRBaddterm](#). If the indicated name is already in use, BCL adds an index to it. If no constraint name is given, BCL generates a default name starting with `CTR`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).)

Related topics

[XPRBnewarrsum](#), [XPRBnewctr](#), [XPRBnewprec](#), [XPRBsetdictionarysize](#).

XPRBnewvar

Purpose

Declare a single variable.

Synopsis

```
XPRBvar XPRBnewvar(XPRBprob prob, int type, const char *name, double bdl,  
double bdu);
```

Arguments

prob	Reference to a problem.
type	The variable type, which may be one of: XPRB_PL continuous; XPRB_BV binary; XPRB_UI general integer; XPRB_PI partial integer; XPRB_SC semi-continuous; XPRB_SI semi-continuous integer.
name	The variable name (of unlimited length). May be <code>NULL</code> if not required.
bdl	The variable's lower bound.
bdu	The variable's upper bound.

Return value

Reference to the new variable if function executed successfully, `NULL` otherwise.

Example

```
XPRBprob prob;  
XPRBvar x1, x2;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 1, 100);  
x2 = XPRBnewvar(prob, XPRB_SC, "klm2", 0, 20);
```

This defines an integer variable `x1`, taking values between 1 and 100, with the name `abc3`, and a semi-continuous variable `x2`, taking the value 0 or values between 1 and 20, with the name `klm2`.

Further information

1. The creation of a variable in BCL involves not only its name but also its type and bounds (which may be infinite, defined by the corresponding Xpress-MP constants). The function returns the BCL reference to the variable (*i.e.* a model variable). If the indicated name is already in use, BCL adds an index to it. If no variable name is given, BCL generates a default name starting with `VAR`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBsetdictionarysize](#).) If a partial integer, semi-continuous, or semi-continuous integer variable is being created, the integer or semi-continuous limit (*i.e.* the lower bound of the continuous part for partial integer and semi-continuous, and of the semi-continuous integer part for semi-continuous integer) is set to the maximum of 1 and `bdl`. This value can be subsequently modified with the function [XPRBsetlim](#).
2. The lower and upper bounds may take values of `-XPRB_INFINITY` and `XPRB_INFINITY` for minus and plus infinity respectively.

Related topics

[XPRBnewarrvar](#), [XPRBsetvartype](#), [XPRBstartarrvar](#), [XPRBsetdictionarysize](#).

XPRBprintarrvar

Purpose

Print out an array of variables.

Synopsis

```
int XPRBprintarrvar(XPRBarrvar av);
```

Argument

`av` Reference to an array of variables.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;
XPRBarrvar ty1;
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "array1", 0, 500);
XPRBprintarrvar(ty1);
```

The above prints names and bounds for all variables in the array `ty1`.

Further information

This function prints out all variables in the array (names and bounds or solution values). It is not available in the student version.

Related topics

[XPRBexportprob](#), [XPRBprintctr](#), [XPRBprintprob](#), [XPRBprintvar](#).

XPRBprintctr

Purpose

Print out a constraint.

Synopsis

```
int XPRBprintctr(XPRBctr ctr);
```

Argument

`ctr` Reference to a constraint.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following prints out the constraint `ctr2`.

```
XPRBprob prob;
XPRBctr ctr2;
XPRBarrvar ty1;
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
ctr2 = XPRBnewsum(prob, "r2", ty1, XPRB_E, 9);
XPRBprintctr(ctr2);
```

Further information

This function prints out a constraint in LP format. It is not available in the student version.

Related topics

[XPRBexportprob](#), [XPRBprintprob](#), [XPRBprintarrvar](#), [XPRBprintvar](#).

XPRBprintcut

Purpose

Print out a cut.

Synopsis

```
int XPRBprintcut(XPRBcut cut);
```

Argument

`cut` Reference to a cut.

Return value

0 if function executed successfully, 1 otherwise.

Example

Print out the cut *cut2*.

```
XPRBcut cut2;
XPRBarrvar ty1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
ty1 = XPRBnewarrvar(expl1, 5, XPRB_PL, "arry1", 0, 500);
cut2 = XPRBnewcutsum(expl1, ty1, XPRB_E, 9, 3);
XPRBprintcut(cut2);
```

Further information

This function prints out a cut in LP-format. It is not available in the Student Edition.

Related topics

[XPRBnewcut](#).

XPRBprintf

Purpose

Print text and other program output.

Synopsis

```
int XPRBprintf(XPRBprob prob, const *format, ...);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>format</code>	String indicating the format of the text to be output. Format parameters are identical to those of the C function <code>printf</code> .
<code>...</code>	Items to be printed according to the format specification in the format string, separated by commas.

Return value

Number of characters printed, or -1 if output truncated.

Example

The following code outputs the string "New variable: abc3", followed by "A real number: 1.3, an integer: 5" on the next line.

```
XPRBprob prob;
XPRBvar x1;
double a=1.3;
int i=5;
...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 1, 100);
XPRBprintf(prob, "New variable: %s\n", XPRBgetvarname(x1));
XPRBprintf(prob, "A real number: %g, an integer: %d", a, i);
```

Further information

This function prints out text, data etc. from the user's program. It behaves like the C function `printf` with the additional feature that whenever the printing callback `XPRBdefcbmsg` is defined, this callback is executed instead of printing to the standard output channel.

Related topics

[XPRBprintprob](#), [XPRBreadlinecb](#).

XPRBprintidxset

Purpose

Print out an index set.

Synopsis

```
int XPRBprintidxset(XPRBidxset idx);
```

Argument

`idx` Reference to an index set.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBidxset iset;  
...  
iset = XPRBnewidxset(prob, "Set", 100);  
XPRBprintidxset(iset);
```

The above prints out the index set `iset`.

Further information

This function prints out an index set. It is not available in the student version.

Related topics

[XPRBprintctr](#), [XPRBprintf](#), [XPRBprintsos](#), [XPRBprintvar](#).

XPRBprintobj

Purpose

Print out the current objective function of a problem.

Synopsis

```
int XPRBprintobj(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following prints out the objective function defined for problem `expl2`.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBprintobj(expl2);
```

Further information

This function prints out the objective function currently defined for the given problem. This function is not available in the student version.

Related topics

[XPRBsetobj](#).

XPRBprintprob

Purpose

Print out the specified problem.

Synopsis

```
int XPRBprintprob(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following prints out the current problem definition.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBprintprob(expl2);
```

Further information

This function prints out the complete problem definition currently held in BCL, that means, the list of constraints, any Special Ordered Sets that have been defined, and the objective function. This function is not available in the student version.

Related topics

[XPRBexportprob](#), [XPRBprintf](#).

XPRBprintsos

Purpose

Print out a Special Ordered Set.

Synopsis

```
int XPRBprintsos(XPRBsos sos);
```

Argument

`sos` Reference to a Special Ordered Set.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBsos set1;  
...  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
XPRBprintsos(set1);
```

This prints out the SOS `set1`.

Further information

This function prints out a Special Ordered Set. It is not available in the student version.

Related topics

[XPRBprintctr](#), [XPRBprintidxset](#), [XPRBprintprob](#), [XPRBprintvar](#).

XPRBprintvar

Purpose

Print out a variable.

Synopsis

```
int XPRBprintvar(XPRBvar var);
```

Argument

`var` BCL reference for a variable.

Return value

Number of characters printed.

Example

The following code outputs `abc3[1.000,100.000]`, followed by `abc4[0.000,5.000,50.000]`.

```
XPRBprob prob;
XPRBvar x1, x3;

...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 1, 100);
XPRBprintvar(x1);
x3 = XPRBnewvar(prob, XPRB_SC, "abc4", 0, 50);
XPRBsetlim(x3, 5);
XPRBprintvar(x3);
```

Further information

This function prints out a variable: name and bounds for continuous, binary and integer variables; name, bounds and integer limit or lower semi-continuous limit for partial integer, semi-continuous, and semi-continuous integer variables; or, where a solution has been computed, name and solution value.

Related topics

[XPRBprintctr](#), [XPRBprintidxset](#), [XPRBprintprob](#), [XPRBprintsos](#).

XPRBreadarrlinecb

Purpose

Read a line of an array from a data file.

Synopsis

```
int XPRBreadarrlinecb(char *(*fgs)(char *,int,void *), void *file,
                      int length, const char *format, void *arrc, int size);
```

Arguments

<code>fgs</code>	The system's <code>fgets</code> function (defined by <code>XPRB_FGETS</code>).
<code>file</code>	Pointer to a data file.
<code>length</code>	Maximum length of any text string to be read.
<code>format</code>	String indicating the format of the data file to be read, consisting of one of the listed values followed by a separator sign: <code>t[num]</code> text up to next separator sign or space (blank/tabulator/line break), optionally followed by the maximum string length to be read; <code>s[num]</code> text marked by single quotes (' '), optionally followed by the maximum string length to be read; <code>S[num]</code> text marked by double quotes (" "), optionally followed by the maximum string length to be read; <code>T[num]</code> text, as for <code>t</code> , <code>s</code> , or <code>S</code> , depending on the first character read, optionally followed by the maximum string length to be read; <code>i</code> integer value; <code>g</code> real (float) value.
<code>arrc</code>	Array of size at least <code>size</code> that receives the data that are read.
<code>size</code>	Maximum number of data items to be read.

Return value

Number of data items read.

Example

```
double vlist[10];
FILE *datafile;
...
datafile=fopen("filename", "r");
XPRBreadlinecb(XPRB_FGETS, datafile, 120, "g ", vlist, 6);
fclose(datafile);
```

This opens a data file and reads a line of six double values separated by spaces, before closing the file.

Further information

This function reads tables from data files in a format compatible with the `diskdata` command of `mp-model` and `Mosel`. Data lines in the input file may be continued over several lines by using the line continuation sign `&`. The input file may also contain comments (preceded by `!`) and empty lines, both of which are skipped over. The data file is accessed with standard C functions (`fopen`, `fclose`). The function reads up to `size` data items of the type indicated by the format parameter. All string types in the format may (optionally) be followed by the maximum string length to be read. Otherwise the maximum length is assumed to be `length`. The type of separator signs (e.g. `,` `;` `:`) used in the data file needs to be given after the format option(s). Array `arrc` is an array of the same type as the data to be read (`int *`, `char *`, or `double *`) and of size at least `size`. With function `XPRBsetdecsign` the decimal sign used in the data input may be changed, for instance to use a decimal comma.

Related topics

`XPRBreadlinecb`, `XPRBsetdecsign`.

XPRBreadlinecb

Purpose

Read a fixed-format line from a data file.

Synopsis

```
int XPRBreadlinecb(char *(*fgs)(char *,int,void *), void *file,
                   int length, const char *format, ...);
```

Arguments

<code>fgs</code>	The system's <code>fgets</code> function (defined by <code>XPRB_FGETS</code>).
<code>file</code>	Pointer to a data file.
<code>length</code>	Maximum length of any text string to be read.
<code>format</code>	String indicating the format of the data line to be read, which may be one of: <code>t[num]</code>] text up to next separator sign or space (blank / tab / line break), optionally followed by the maximum string length to be read; <code>s[num]</code>] text marked by single quotes, ' ', optionally followed by the maximum string length to be read; <code>S[num]</code>] text marked by double quotes, " ", optionally followed by the maximum string length to be read; <code>T[num]</code>] text as for <code>t</code> , <code>s</code> , or <code>S</code> , depending on the first character read, optionally followed by the maximum string length to be read; <code>i</code> integer value; <code>g</code> real (float) value. The number of format parameters is arbitrary.
<code>...</code>	Addresses of items that are to be read, separated by commas.

Return value

Number of data items read.

Example

The following opens a data file for reading, reads a line with text and a double value, separated by a semi-colon, and then reads a line with two integers and a string of up to ten characters marked by single quotes, all separated by blanks, before finally closing the file.

```
double value;
FILE *datafile;
char name[100];
int i[2];
...
datafile=fopen("filename", "r");
XPRBreadlinecb(XPRB_FGETS, datafile, 99, "T;g", name, &value);
XPRBreadlinecb(XPRB_FGETS, datafile, 50, "i i s[10]", &i[0],
               &i[1], name);
fclose(datafile);
```

Further information

This function reads input data files in a format compatible with the `diskdata` command of `mp-model` and `Mosel`. Data lines in the input file may be continued over several lines by using the line continuation sign `&`. The input file may also contain comments (preceded by `!`) and empty lines, both of which are skipped over. The data file is accessed with standard C functions (`fopen`, `fclose`). The format string gives the type of data item to be read. Each string type may (optionally) be followed by the maximum length to be read. Otherwise, the maximum length is assumed to be `length`. The type of separator signs (e.g. `,` `;` `:`) used in the data file needs to be indicated between each pair of format options. As with the C functions `printf` or `scanf`, the format string is followed by the addresses where the data are stored. With function `XPRBsetdecsign` the decimal sign used in the data input may be changed, for instance to use a decimal comma.

Related topics

`XPRBreadarrlinecb`, `XPRBsetdecsign`.

XPRBresetprob

Purpose

Release system resources used for storing solution information.

Synopsis

```
int XPRBresetprob(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following resets and frees resources used by BCL and Xpress-Optimizer for storing solution information:

```
XPRBprob expl2;  
expl2 = XPRBnewprob(NULL);  
...  
XPRBsolve(expl2, "");  
...  
XPRBresetprob(expl2);
```

Further information

This function deletes any solution information stored in BCL; it also deletes the corresponding Xpress-Optimizer problem and removes any auxiliary files that may have been created by optimization runs. It also resets the Optimizer control parameters for sparse matrix elements (`EXTRACOLS`, `EXTRAROWS`, and `EXTRAELEMS`) to their default values. The BCL problem definition itself remains. This function may be used to free up memory if the solution information is not required any longer but the problem definition is to be kept for later (re)use. To completely delete a problem the function `XPRBdelprob` needs to be used.

Related topics

`XPRBdelprob`, `XPRBfinish`.

XPRBsavebasis

Purpose

Save the current basis.

Synopsis

```
XPRBbasis XPRBsavebasis(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

Reference to the saved basis.

Example

```
XPRBprob expl2;  
XPRBbasis basis;  
expl2 = XPRBnewprob("example2");  
...  
XPRBsolve(expl2, "l");  
basis = XPRBsavebasis(expl2);
```

This saves the current basis.

Further information

This function saves the current basis of a problem. The basis may be reinput using function [XPRBloadbasis](#). These two functions serve for storing bases in memory; for writing a basis to a file, the Optimizer library function [XPRSwritebasis](#) may be used. Note that there is no need to allocate space for the basis, but after its use, the basis should be deleted using function [XPRBdelbasis](#). You may have to switch linear presolve and integer preprocessing off (Optimizer library controls `PRESOLVE` and `MIPPRESOLVE`) in order for the saving and reloading of bases to work correctly.

Related topics

[XPRBdelbasis](#), [XPRBloadbasis](#), [XPRSreadbasis](#) (see Optimizer Reference Manual), [XPRSwritebasis](#) (see Optimizer Reference Manual).

XPRBsetarrvarel

Purpose

Add an entry to a variable array in a given position.

Synopsis

```
int XPRBsetarrvarel(XPRBarrvar av, int ndx, XPRBvar var);
```

Arguments

<code>av</code>	BCL reference to an array.
<code>ndx</code>	Index within the array.
<code>var</code>	Variable to be added to the array.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBarrvar av2;  
XPRBvar x1;  
  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
av2 = XPRBstartarrvar(prob, 5, "arr2");  
XPRBsetarrvarel(av2, 3, x1);
```

This inserts variable `x1` at the fourth position of the array `av2` (which is numbered from 0).

Further information

This function puts a variable in specified position within the array. If there is already a variable at this position it is overwritten.

Related topics

[XPRBapparrvarel](#), [XPRBdelarrvar](#), [XPRBendarrvar](#), [XPRBnewarrvar](#), [XPRBstartarrvar](#).

XPRBsetcolorder

Purpose

Set a column ordering criterion for matrix generation.

Synopsis

```
int XPRBsetcolorder(XPRBprob prob, int num);
```

Arguments

prob	Reference to a problem.
num	The ordering flag, which must be one of:
0	default ordering;
1	alphabetical order.

Return value

0 if function executed successfully, 1 otherwise.

Example

Set a fixed ordering for a single problem:

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
XPRBsetcolorder(expl2, 1);
```

Further information

1. BCL runs reproduce always the same matrix for a problem. This function allows the user to choose a different ordering criterion than the default one. Note that this function only changes the order of columns in what is sent to Xpress-Optimizer, you do not see any effect when exporting the matrix with BCL. However you can control the effect by exporting the matrix from the Optimizer.
2. This function can be used before any problem has been created (with first argument `NULL`). In this case the setting applies to all problems that are created subsequently.

Related topics

[XPRBloadmat](#), [XPRBnewprob](#).

XPRBsetctrtype

Purpose

Set the constraint type.

Synopsis

```
int XPRBsetctrtype(XPRBctr ctr, int qrtype);
```

Arguments

<code>ctr</code>	Reference to a previously created constraint.
<code>qrtype</code>	The constraint type, which must be one of: <code>XPRB_L</code> 'less than or equal to' constraint; <code>XPRB_G</code> 'greater than or equal to' constraint; <code>XPRB_E</code> an equality; <code>XPRB_N</code> a non-binding row (objective function).

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBctr ctrl;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);  
XPRBsetctrtype(ctrl, XPRB_L);
```

This changes `ctrl` to a 'less than or equal to' constraint.

Further information

This function changes the type of a previously defined constraint to inequality, equation or non-binding. Function [XPRBsetrange](#) has to be used for changing the constraint to a ranged constraint. If a ranged constraint is changed back to some other type with this function, its upper bound becomes the right hand side value.

Related topics

[XPRBgetctrtype](#), [XPRBnewctr](#), [XPRBsetrange](#), [XPRBsetterm](#).

XPRBsetcutid

Purpose

Set the classification or identification number of a cut.

Synopsis

```
int XPRBsetcutid(XPRBcut cut, int id);
```

Arguments

<code>cut</code>	Reference to a previously created cut.
<code>id</code>	Classification or identification number.

Return value

0 if function executed successfully, 1 otherwise.

Example

Set the classification or identification number of the cut *cut1* to 10.

```
XPRBcut cut1;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
XPRBsetcutid(cut1, 10);
```

Further information

This function changes the classification or identification number of a previously defined cut. This change does not have any effect on the cut definition in Xpress-Optimizer if the cut has already been added to the matrix with the function [XPRBaddcuts](#).

Related topics

[XPRBnewcut](#), [XPRBgetcutid](#), [XPRBsetcuttype](#).

XPRBsetcutmode

Purpose

Set the cut mode.

Synopsis

```
int XPRBsetcutmode(XPRBprob prob, int mode);
```

Arguments

prob	Reference to a problem.
mode	Cut mode indicator:
0	switch cut mode off
1	switch cut mode on

Return value

0 if function executed successfully, 1 otherwise.

Example

The example shows how to enable the cut mode.

```
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
XPRBsetcutmode(expl1, 1);
```

Further information

This function switches the cut mode on or off. It changes the settings of certain Optimizer controls. Switching the cut mode off resets these controls to their default values.

Related topics

[XPRBaddcuts](#).

XPRBsetcutterm

Purpose

Set a cut term.

Synopsis

```
int XPRBsetcutterm(XPRBcut cut, XPRBvar var, double coeff);
```

Arguments

`cut` Reference to a previously created cut.
`var` Reference to a variable, may be `NULL`.
`coeff` Value of the coefficient of the variable `var`.

Return value

0 if function executed successfully, 1 otherwise.

Example

Set the RHS of the cut *cut1* to 7.0.

```
XPRBcut cut1;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
XPRBsetcutterm(cut1, NULL, 7.0);
```

Further information

This function sets the coefficient of a variable to the value `coeff`. If `var` is set to `NULL`, the right hand side of the cut is set to `coeff`.

Related topics

[XPRBnewcut](#), [XPRBaddcutterm](#), [XPRBdelcutterm](#).

XPRBsetcuttype

Purpose

Set the type of a cut.

Synopsis

```
int XPRBsetcuttype(XPRBcut cut, int type);
```

Arguments

<code>cut</code>	Reference to a previously created cut.
<code>type</code>	Type of the cut: XPRB_L \leq (inequality) XPRB_G \geq (inequality) XPRB_E = (equation)

Return value

0 if function executed successfully, 1 otherwise.

Example

Set the type of *cut1* to ' \leq '.

```
XPRBcut cut1;  
XPRBprob expl1;  
expl1 = XPRBnewprob("cutexample");  
cut1 = XPRBnewcut(expl1, XPRB_E, 1);  
XPRBsetcuttype(cut1, XPRB_L);
```

Further information

This function changes the type of the given cut. This change does not have any effect on the cut definition in Xpress-Optimizer if the cut has already been added to the matrix with the function [XPRBaddcuts](#).

Related topics

[XPRBnewcut](#), [XPRBgetcuttype](#), [XPRBgetcutid](#).

XPRBsetdecsign

Purpose

Select the decimal sign for data input.

Synopsis

```
int XPRBsetdecsign(char sign);
```

Argument

`sign` The decimal sign to be used. This is typically '.' (default), or ','.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBsetdecsign(',');
```

This switches to using a comma as the decimal point.

Further information

By default, BCL uses the Anglo-American decimal point when reading and writing real numbers. With this function the decimal sign accepted by the data input functions [XPRBreadlinecb](#) and [XPRBreadarrlinecb](#) can be changed to a comma or any other non-numerical ASCII character. Note that all output still contains the decimal point.

Related topics

[XPRBreadarrlinecb](#), [XPRBreadlinecb](#).

XPRBsetdictionarysize

Purpose

Set the size of a dictionary.

Synopsis

```
int XPRBsetdictionarysize(XPRBprob prob, int dict, int size)
```

Arguments

<code>prob</code>	Reference to a problem.
<code>dict</code>	Choice of the dictionary. Possible values: <code>XPRB_DICT_NAMES</code> names dictionary <code>XPRB_DICT_IDX</code> indices dictionary
<code>size</code>	Non-negative value, preferably a prime number; 0 disables the dictionary (for names dictionary only).

Return value

0 if function executed successfully, 1 otherwise.

Example

Switch off the names dictionary:

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
XPRBsetdictsize(expl2, XPRB_DICT_NAMES, 0);
```

Further information

1. This function sets the size of the hash table of the names or indices dictionaries (defaults: names 2999, indices 1009) of the given problem. It can only be called immediately after the creation of the corresponding problem.
2. The names dictionary serves for storing and accessing the names of all modeling objects (variables, arrays of variables, constraints, SOS, index sets). Once it has been disabled it cannot be enabled any more. All methods relative to the names cannot be used if this dictionary has been disabled and BCL will not generate any unique names at the creation of model objects. If you wish to use the names dictionary we recommend to choose a size close to the number of variables+constraints in your problem, preferably a prime number.
3. The indices dictionary serves for storing all index set elements. The indices dictionary cannot be disabled, it is created automatically once an index set element is defined.

Related topics

[XPRBnewprob](#), [XPRBgetbyname](#).

XPRBseterrctrl

Purpose

Select behavior in case of an error.

Synopsis

```
int XPRBseterrctrl(int flag)
```

Argument

flag	Indicator value for error handling. May be one of:
0	no error handling by BCL;
1	program exit at error (default).

Return value

0 if function executed successfully, 1 otherwise.

Example

The following switches to error handling by the user's own program.

```
XPRBseterrctrl(0);
```

Further information

1. This function controls whether BCL performs error handling. By default, the execution is stopped whenever an error occurs. If the error handling by BCL is disabled, the user needs to perform the checking for errors in his program by testing the return values of all functions or using the callback function [XPRBdefcberr](#). It may be preferable to disable the error handling by BCL if a BCL program is embedded into some larger application or executed under Windows. Callback function [XPRBdefcberr](#) can be defined to retrieve the error messages and implement user error handling.
2. This function can be used before BCL has been initialized.

Related topics

[XPRBdefcberr](#), [XPRBgetversion](#).

XPRBsetlb

Purpose

Set a lower bound.

Synopsis

```
int XPRBsetlb(XPRBvar var, double bdl);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>bdl</code>	The variable's new lower bound.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code changes the lower bound of `x1` to 3.

```
XPRBprob prob;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 1, 100);  
XPRBsetlb(x1, 3.0);
```

Further information

This function sets the lower bound on a variable.

Related topics

[XPRBfixvar](#), [XPRBgetbounds](#), [XPRBgetlim](#), [XPRBsetlim](#), [XPRBsetub](#).

XPRBsetlim

Purpose

Set the integer limit for a partial integer, or the lower semi-continuous limit for a semi-continuous or semi-continuous integer variable.

Synopsis

```
int XPRBsetlim(XPRBvar var, double c);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>c</code>	Value of the integer limit.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBvar x3;  
...  
x3 = XPRBnewvar(prob, XPRB_SC, "abc4", 0, 50);  
XPRBsetlim(x3, 5);
```

This sets the limit for variable `x3` to 5. The possible values for `x3` are thus reduced from `x3 = 0 or 1 <= x3 <= 50` at the creation of this variable to `x3 = 0 or 5 <= x3 <= 50`.

Further information

This function sets the integer limit (*i.e.* the lower bound of the continuous part) of a partial integer variable or the semi-continuous limit of a semi-continuous or semi-continuous integer variable to the given value.

Related topics

[XPRBfixvar](#), [XPRBgetbounds](#), [XPRBgetlim](#), [XPRBsetlb](#), [XPRBsetub](#).

XPRBsetmodcut

Purpose

Set the constraint type.

Synopsis

```
int XPRBsetmodcut(XPRBctr ctr, int mcstat);
```

Arguments

<code>ctr</code>	Reference to a previously created constraint.
<code>mcstat</code>	The constraint type, which must be one of:
0	constraint;
1	model cut.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following turns the constraint `ctrl` into a model cut.

```
XPRBprob prob;  
XPRBctr ctrl;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);  
XPRBsetmodcut(ctrl, 1);
```

Further information

This function changes the type of a previously defined constraint from ordinary constraint to model cut and vice versa.

Related topics

[XPRBdelctr](#), [XPRBgetmodcut](#), [XPRBnewctr](#).

XPRBsetmsglevel

Purpose

Set the message print level.

Synopsis

```
int XPRBsetmsglevel(XPRBprob prob, int level);
```

Arguments

prob	Reference to a problem.
level	The message level, <i>i.e.</i> the type of messages printed by BCL. This may be one of: 0 no messages printed; 1 error messages only printed; 2 warnings and errors printed; 3 warnings, errors, and Optimizer log printed (default); 4 all messages printed.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following statement switches to printing error messages only.

```
XPRBprob prob;  
...  
XPRBsetmsglevel(prob, 1);
```

Further information

1. BCL can produce different types of messages; status information, warnings and errors. This function controls which of these are output. For settings 1 or higher, the corresponding Optimizer output is also displayed. In addition to this setting, the amount of Optimizer output can be modified through several Optimizer printing control parameters (see the 'Xpress-Optimizer Reference Manual').
2. This function may be used before any problem has been created and even before BCL has been initialized (with first argument NULL). In this case the setting applies to all problems that are created subsequently.

Related topics

[XPRBdefcbmsg.](#)

XPRBsetobj

Purpose

Select the objective function.

Synopsis

```
int XPRBsetobj(XPRBprob prob, XPRBctr ctr);
```

Arguments

`prob` Reference to a problem.
`ctr` Reference to a previously defined constraint.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBctr ctr3;  
XPRBarrvar tobj;  
...  
tobj = XPRBnewarrvar(prob, 10, XPRB_PL, "tabo", 0, 800);  
ctr3 = XPRBnewsum(prob, "r3", tobj, XPRB_N, 0);  
XPRBsetobj(prob, ctr3);
```

This defines a non-binding constraint, `ctr3`, and then sets it as the objective function.

Further information

This functions sets the objective function by selecting a constraint the variable terms of which become the objective function. This must be done before any optimization task is carried out. Typically, the objective constraint will have the type `XPRB_N` (non-binding), but any other type of constraint may be chosen too. In the latter case, the equation or inequality expressed by the constraint also remains part of the problem.

Related topics

[`XPRBgetsense`](#), [`XPRBsetsense`](#).

XPRBsetqterm

Purpose

Set a quadratic constraint term.

Synopsis

```
int XPRBsetqterm(XPRBctr ctr, XPRBvar var1, XPRBvar var2,
                 double coeff);
```

Arguments

<code>ctr</code>	Reference to a previously defined constraint.
<code>var1</code>	Reference to a variable.
<code>var2</code>	Reference to a variable (not necessarily different).
<code>coeff</code>	Value to be added to the coefficient of the term <code>var1 * var2</code> .

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;
XPRBvar x2;
XPRBctr ctrl;
...
ctrl = XPRBnewctr(prob, "r1", XPRB_L);
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
XPRBaddqterm(ctrl, x2, x2, 1);
XPRBsetqterm(ctrl, x2, x2, 5.2);
```

This sets the coefficient of the term `x2*x2` to 5.2.

Further information

This function sets the coefficient of a quadratic term in a constraint to the value `coeff`.

Related topics

[XPRBaddqterm](#), [XPRBdelqterm](#).

XPRBsetrange

Purpose

Define a range constraint.

Synopsis

```
int XPRBsetrange(XPRBctr ctr, double bdl, double bdu);
```

Arguments

<code>ctr</code>	Reference to the constraint.
<code>bdl</code>	Lower bound on the range constraint.
<code>bdu</code>	Upper bound on the range constraint.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following transforms the equality constraint `ctr2` into the ranged constraint `4.0 <= sum(i=0:4) ty1[i] <= 15.5`.

```
XPRBprob prob;
XPRBctr ctr2;
XPRBarrvar ty1;
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
ctr2 = XPRBnewsum(prob, "r2", ty1, XPRB_E, 9);
XPRBsetrange(ctr2, 4.0, 15.5);
```

Further information

This function changes the type of a previously defined constraint to a range constraint within the bounds specified by `bdl` and `bdu`. The constraint type and right hand side value of the constraint are replaced by the type `XPRB_R` (range) and the two bounds.

Related topics

[XPRBgetctrtype](#), [XPRBgetrange](#), [XPRBsetctrtype](#).

XPRBsetrealfmt

Purpose

Set the format for printing real numbers.

Synopsis

```
int XPRBsetrealfmt(XPRBprob prob, const char *fmt);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>fmt</code>	Format string (as used by the C function <code>printf</code>). Simple format strings are of the form <code>%n</code> where <code>n</code> may be, for instance, one of
<code>g</code>	default printing format (precision: 6 digits; exponential notation if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision)
<code>.numf</code>	print real numbers in the style <code>[-]d.d</code> where the number of digits after the decimal point is equal to the given precision <code>num</code> .

Return value

0 if function executed successfully, 1 otherwise.

Example

This example sets the number printing format to 10 digits after the decimal point:

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
XPRBsetrealfmt(expl2, "%.10f");
```

Further information

1. In problems with very large or very small numbers it may become necessary to change the printing format to obtain a more exact output. In particular, by changing the precision it is possible to reduce the difference between matrices loaded in memory into Xpress-Optimizer and the output produced by exporting them to a file.
2. This function can be used before any problem has been created (with first argument `NULL`). In this case the setting applies to all problems that are created subsequently.

Related topics

[XPRBexportprob](#), [XPRBloadmat](#), [XPRBprintprob](#).

XPRBsetsense

Purpose

Set the sense of the objective function.

Synopsis

```
int XPRBsetsense(XPRBprob prob, int dir);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>dir</code>	Sense of the objective function, which must be one of: <code>XPRB_MAXIM</code> maximize the objective; <code>XPRB_MINIM</code> minimize the objective.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob expl2;  
...  
expl2 = XPRBnewprob("example2");  
XPRBsetsense(expl2, XPRB_MAXIM);
```

This sets `expl2` as a maximization problem.

Further information

This functions sets the objective sense to maximization or minimization. It is set to minimization by default.

Related topics

[XPRBgetsense](#), [XPRBsetobj](#).

XPRBsetsosdir

Purpose

Set a branching directive for a SOS.

Synopsis

```
int XPRBsetsosdir(XPRBsos sos, int type, double val);
```

Arguments

<code>sos</code>	Reference to a previously created SOS.
<code>type</code>	The directive type, which must be one of: <code>XPRB_PR</code> priority; <code>XPRB_UP</code> first branch upwards; <code>XPRB_DN</code> first branch downwards; <code>XPRB_PU</code> pseudo cost on branching upwards; <code>XPRB_PD</code> pseudo cost on branching downwards.
<code>val</code>	An argument dependent on the type of the directive being defined. If <code>type</code> is: <code>XPRB_PR</code> <code>val</code> will be the priority value, an integer between 1 (highest) and 1000 (lowest), the default; <code>XPRB_UP</code> no input is required — choose any value, e.g. 0; <code>XPRB_DN</code> no input is required — choose any value, e.g. 0; <code>XPRB_PU</code> <code>val</code> will be the value of the pseudo cost for the upward branch; <code>XPRB_PD</code> <code>val</code> will be the value of the pseudo cost for the downward branch.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBsos set1;  
  
...  
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);  
XPRBsetsosdir(set1, 5);  
XPRBsetsosdir(set1, XPRB_DN, 0);
```

This gives a priority of 5 to the SOS `set1` and sets branching downwards as the preferred direction for `set1`.

Further information

This function sets any type of branching directive available in Xpress-MP. This may be a priority for branching on a SOS (type `XPRB_PR`), the preferred branching direction (types `XPRB_UP`, `XPRB_DN`) or the estimated cost incurred when branching on a SOS (types `XPRB_PU`, `XPRB_PD`). Several directives of different types may be set for a single set. Function `XPRBsetvardir` may be used to set a directive for a variable.

Related topics

`XPRBcleardir`, `XPRBsetvardir`.

XPRBsetterm

Purpose

Set a linear constraint term.

Synopsis

```
int XPRBsetterm(XPRBctr ctr, XPRBvar var, double coeff);
```

Arguments

ctr BCL reference to a previously created constraint.
var BCL reference to a variable. May be `NULL` if not required.
coeff Value of the coefficient of the variable `var`.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;  
XPRBctr ctrl;  
...  
ctrl = XPRBnewctr(prob, "r1", XPRB_E);  
XPRBsetterm(ctrl, NULL, 7.0);
```

This sets the right hand side of the constraint `ctrl` to 7.0.

Further information

This function sets the coefficient of a variable to the value `coeff`. If `var` is set to `NULL`, the right hand side of the constraint is set to `coeff`.

Related topics

[XPRBaddterm](#), [XPRBdelctr](#), [XPRBnewctr](#).

XPRBsetub

Purpose

Set an upper bound.

Synopsis

```
int XPRBsetub(XPRBvar var, double bdu);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>bdu</code>	The variable's new upper bound.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code changes the upper bound of `x1` to 200.

```
XPRBprob prob;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 1, 100);  
XPRBsetub(x1, 200.0);
```

Further information

This function sets the upper bound on a variable.

Related topics

[XPRBfixvar](#), [XPRBgetbounds](#), [XPRBgetlim](#), [XPRBsetlb](#), [XPRBsetlim](#).

XPRBsetvardir

Purpose

Set a branching directive for a variable.

Synopsis

```
int XPRBsetvardir(XPRBvar var, int type, double c);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>type</code>	Directive type, which must be one of: <code>XPRB_PR</code> priority; <code>XPRB_UP</code> first branch upwards; <code>XPRB_DN</code> first branch downwards; <code>XPRB_PU</code> pseudo cost on branching upwards; <code>XPRB_PD</code> pseudo cost on branching downwards.
<code>c</code>	An argument dependent on the type of directive to be defined. Must be one of: <code>XPRB_PR</code> priority value — an integer between 1 (highest) and 1000 (least priority), the default; <code>XPRB_UP</code> no input required — set to any value, e.g. 0; <code>XPRB_DN</code> no input required — set to any value, e.g. 0; <code>XPRB_PU</code> value of the pseudo cost on branching upwards; <code>XPRB_PD</code> value of the pseudo cost on branching downwards.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following example gives a priority of 10 to variable `x1` and sets the preferred branching direction to be upwards.

```
XPRBprob prob;  
XPRBvar x1;  
  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
XPRBsetvardir(x1, XPRB_PR, 10);  
XPRBsetvardir(x1, XPRB_UP, 0);
```

Further information

1. This function sets any type of branching directive available in Xpress-MP. This may be a priority for branching on a variable (type `XPRB_PR`), the preferred branching direction (types `XPRB_UP`, `XPRB_DN`) or the estimated cost incurred when branching on a variable (types `XPRB_PU`, `XPRB_PD`). Several directives of different types may be set for a single variable.
2. Note that it is only possible to set branching directives for discrete variables (including semi-continuous and partial integer variables). Function `XPRBsetsosdir` may be used to set a directive for a SOS.

Related topics

`XPRBcleardir`, `XPRBsetsosdir`.

XPRBsetvarlink

Purpose

Set the interface pointer of a variable.

Synopsis

```
int XPRBsetvarlink(XPRBvar var, void *link);
```

Arguments

<code>var</code>	Reference to a BCL variable
<code>link</code>	Pointer to an interface object

Return value

0 if function executed successfully, 1 otherwise.

Example

Set the interface pointer of variable `x1` to `vlink`:

```
XPRBprob prob;  
XPRBvar x1;  
myinterfacetype *vlink;  
...  
x1 = XPRBnewvar(prob, XB_UI, "abc3", 0, 100);  
XPRBsetvarlink(x1, vlink);
```

Further information

This function sets the interface pointer of a variable to the indicated object. It may be used to establish a connection between a variable in BCL and some other external program.

Related topics

[XPRBgetvarlink](#), [XPRBdefcbdelvar](#).

XPRBsetvartype

Purpose

Set the variable type.

Synopsis

```
int XPRBsetvartype(XPRBvar var, int type);
```

Arguments

<code>var</code>	BCL reference to a variable.
<code>type</code>	The variable type, which is one of: <code>XPRB_PL</code> continuous; <code>XPRB_BV</code> binary; <code>XPRB_UI</code> general integer; <code>XPRB_PI</code> partial integer; <code>XPRB_SC</code> semi-continuous; <code>XPRB_SI</code> semi-continuous integer.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code changes the type of variable `x1` from integer to binary, and consequently reducing the upper bound to 1.

```
XPRBprob prob;  
XPRBvar x1;  
...  
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);  
XPRBsetvartype(x1, XPRB_BV);
```

Further information

This function changes the type of a variable that has been created previously.

Related topics

[XPRBgetvarname](#), [XPRBgetvartype](#), [XPRBnewvar](#).

XPRBsolve

Purpose

Call the Xpress-Optimizer solution algorithm.

Synopsis

```
int XPRBsolve(XPRBprob prob, char *alg);
```

Arguments

prob	Reference to a problem.
alg	Choice of the solution algorithm, which should be one of: <ul style="list-style-type: none">" " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);"d" solve the problem using the dual simplex algorithm;"p" solve the problem using the primal simplex algorithm;"b" solve the problem using the Newton barrier algorithm;"n" use the network solver (LP only);"l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);"g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code uses the primal simplex algorithm to solve `expl2` as a MIP, assuming that it contains global entities.

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBsolve(expl2, "pg");
```

Further information

This function selects and starts the Xpress-Optimizer solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, e.g. "dg". If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition it is regenerated automatically prior to the start of the algorithm. Matrix reloading can also be forced by calling `XPRBsync` before the optimization. The sense of the optimization (default: minimization) can be changed with function `XPRBsetsense`. Before solving a problem, the objective function must be selected with `XPRBsetobj`. Note that if you use an incomplete global search you should finish your program with a call to the Optimizer library function `XPRSinitglobal` in order to remove all search tree information that has been stored. Otherwise you may not be able to re-run your program.

Related topics

`XPRBgetsense`, `XPRBmaxim`, `XPRBminim`, `XPRBsetsense`, `XPRBsync`.

XPRBstartarrvar

Purpose

Start the definition of a variable array.

Synopsis

```
XPRBarrvar XPRBstartarrvar(XPRBprob prob, int nbvar, const char *name);
```

Arguments

<code>prob</code>	Reference to a problem.
<code>nbvar</code>	The maximum number of variables in the array.
<code>name</code>	Name of the array. May be <code>NULL</code> if not required.

Return value

Reference to the new array if function executed successfully, `NULL` otherwise.

Example

```
XPRBprob prob;  
XPRBarrvar av2;  
...  
av2 = XPRBstartarrvar(prob, 5, "arr2");
```

This starts the definition of an array with five elements, named `arr2`.

Further information

This function starts the definition of a variable array. It returns a reference to an array of variables that may be used, for instance, in the definition of constraints. Variables belonging to an array created by this function may stem from any LP-variables previously defined. They may be of different types, and can be positioned in any order. A variable may belong to several arrays, but it is created only once (functions [XPRBnewvar](#) or [XPRBnewarrvar](#)). If the indicated name is already in use, BCL adds an index to it. If no array name is given, BCL generates a default name starting with `AV`.

Related topics

[XPRBdelarrvar](#), [XPRBendarrvar](#), [XPRBnewarrvar](#).

XPRBsync

Purpose

Synchronize BCL with the Optimizer.

Synopsis

```
int XPRBsync(XPRBprob prob, int synctype);
```

Arguments

prob	Reference to a problem.
synctype	Type of the synchronization. Possible values: XPRB_XPRS_SOL update the BCL solution information with the solution currently held in the Optimizer; XPRB_XPRS_PROB force problem reloading.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following forces BCL to reload the matrix into the Optimizer even if there has been no change other than bound changes to the problem definition in BCL since the preceding optimization:

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBsolve(expl2, "l");  
...  
XPRBsync(expl2, XPRB_XPRS_PROB);  
XPRBsolve(expl2, "g");
```

Further information

1. This function resets the BCL problem status.
2. XPRB_XPRS_SOL: at the next solution access the solution information in BCL is updated with the solution held in the Optimizer (after MIP search: best integer solution, otherwise solution of the last LP solved).
3. XPRB_XPRS_PROB: at the next call to optimization or [XPRBloadmat](#) the problem is completely reloaded into the Optimizer; bound changes are not passed on to the problem loaded in the Optimizer any longer.

Related topics

[XPRBgetsol](#), [XPRBgetrcost](#), [XPRBgetdual](#), [XPRBgetslack](#), [XPRBloadmat](#), [XPRBminim](#), [XPRBmaxim](#), [XPRBsolve](#).

XPRBwritedir

Purpose

Write directives to a file.

Synopsis

```
int XPRBwritedir(XPRBprob prob, const char *fname);
```

Arguments

`prob` Reference to a problem.
`fname` Name of the directives files. May be `NULL` if the problem name is to be used.

Return value

0 if function executed successfully, 1 otherwise.

Example

This example writes all directives defined for the problem `expl2` to the file `example2.dir`:

```
XPRBprob expl2;  
expl2 = XPRBnewprob("example2");  
...  
XPRBwritedir(expl2, NULL);
```

Further information

This function writes out to a file the directives defined for a problem. The extension `.dir` is appended to the given file name. When no file name is given, the name of the problem is used. If a file of the given name exists already it is replaced.

Related topics

[XPRBexportprob](#), [XPRBsetvardir](#), [XPRBsetsosdir](#).

Chapter 5

BCL in C++

5.1 An overview of BCL in C++

The C++ interface of BCL provides the full functionality of the C version except for the data input, output and error handling for which the corresponding C functions may be used. The C modeling objects, such as variables, constraints and problems, are converted into classes, and their associated functions into methods of the corresponding class in C++.

To use the C++ version of BCL, the C++ header file must be included at the beginning of the program (and not the main BCL header file `xprb.h`).

```
#include "xprb_cpp.h"
```

Using C++, the termwise definition of constraints is even easier. This has been achieved by overloading the algebraic operators like '+', '-', '<=', or '=='. With these operators constraints may be written in a form that is close to an algebraic formulation.

It should be noted that the names of classes and methods have been adapted to C++ naming standards: All C++ classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsos`, `XPRBindexSet`, `XPRBbasis`) take the same names, with the exception of `XPRBindexSet`. In the names of the methods the prefix `XPRB` has been dropped, as have been references to the type of the object. For example, function `XPRBgetvarname` is turned into the method `getName` of class `XPRBvar`.

All C++ classes of BCL are part of the namespace `dashoptimization`. To use the (short) class names, it is recommended to add the line

```
using namespace ::dashoptimization;
```

at the beginning of every program that uses the C++ classes of BCL.

C++ functions can be used together with C functions, for instance when printing program output or using Xpress-Optimizer functions. However, it is not possible to mix BCL C and C++ objects in a program.

5.1.1 Example

An example of use of BCL in C++ is the following, which constructs the scheduling example described in Chapter 2:

```
#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define NJ      4           // Number of jobs
#define NT     10          // Time limit
```



```

double DUR[] = {3,4,2,2};          // Durations of jobs

XPRBvar start[NJ];                 // Start times of jobs
XPRBvar delta[NJ][NT];            // Binaries for start times
XPRBvar z;                         // Max. completion time

XPRBprob p("Jobs");               // Initialize BCL & a new problem

void jobsModel()
{
    XPRBexpr le;
    int j,t;

    // Create start time variables
    for(j=0;j<NJ;j++) start[j] = p.newVar("start");
    z = p.newVar("z",XPRB_PL,0,NT); // Makespan variable

    // Binaries for each job
    for(j=0;j<NJ;j++)
        for(t=0;t<(NT-DUR[j]+1);t++)
            delta[j][t] =
                p.newVar(XPRBnewname("delta%d%d",j+1,t+1),XPRB_BV);

    // Calculate max. completion time
    p.newCtr("Makespan", start[j]+DUR[j] <= z);
    // Precedence relation betw. jobs
    p.newCtr("Prec", start[0]+DUR[0] <= start[2]);

    // Linking start times & binaries
    {
        le=0;
        for(t=0;t<(NT-DUR[j]+1);t++) le += (t+1)*delta[j][t];
        p.newCtr(XPRBnewname("Link_%d",j+1), le == start[j]);
    }

    // Unique start time for each job
    {
        le=0;
        for(t=0;t<(NT-DUR[j]+1);t++) le += delta[j][t];
        p.newCtr(XPRBnewname("One_%d",j+1), le == 1);
    }

    p.setObj(z);                   // Define and set objective

    for(j=0;j<NJ;j++) start[j].setUB(NT-DUR[j]+1);
    // Upper bounds on "start" var.s
}

void jobsSolve()
{
    int j,t,statmip;

    for(j=0;j<NJ;j++)
        for(t=0;t<NT-DUR[j]+1;t++)
            delta[j][t].setDir(XPRB_PR,10*(t+1));
    // Give highest priority to var.s for earlier start times

    p.setSense(XPRB_MINIM);
    p.solve("g");                  // Solve the problem as MIP
    statmip = p.getMIPStat();       // Get the MIP problem status
    if((statmip == XPRB_MIP_SOLUTION) ||
        (statmip == XPRB_MIP_OPTIMAL))
    {
        // An integer solution has been found
        cout << "Objective: " << p.getObjVal() << endl;
        for(j=0;j<NJ;j++)
        {
            // Print the solution for all start times
            cout << start[j].getName() << ": " << start[j].getSol();
            cout << endl;
        }
    }
}

int main(int argc, char **argv)
{
    jobsModel();                  // Problem definition
    jobsSolve();                  // Solve and print solution
}

```

```

    return 0;
}

```

The definition of SOS is similar to the definition of constraints.

```

XPRBsos set[NJ];

void jobsModel()
{
    ...
    for(j=0; j<NJ; j++)          // Variables for each job
        for(t=0; t<(NT-DUR[j]+1); t++)
            delta[j][t] =
                p.newVar(XPRBnewname("delta%d%d", j+1, t+1), XPRB_PL, 0, 1);

    for(j=0; j<NJ; j++)          // SOS definition
    {
        le=0;
        for(t=0; t<(NT-DUR[j]+1); t++) le += (t+1)*delta[j][t];
        set[j] = p.newSos("sosj", XPRB_S1, le);
    }
}

```

Branching directives for the SOSs are added as follows.

```

for(j=0; j<NJ; j++) set[j].setDir(XPRB_DN);
// First branch downwards on sets

```

Adding the following two lines during or after the problem definition will print the problem to the standard output and export the matrix to a file respectively.

```

p.print();          // Print out the problem def.
p.exportProb(XPRB_MPS, "expl1");
// Output matrix to MPS file

```

Similarly to what has been shown for the problem formulation in C, we may read data from file and use index sets in the problem formulation. The following changes and additions to the basic model formulation are required for the creation of index sets based on data input from file. The function `jobsSolve` is left out in this listing since it remains unchanged from the previous one.

```

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define MAXNJ 4          // Max. number of jobs
#define NT 10           // Time limit

int NJ = 0;             // Number of jobs read in
double DUR[MAXNJ];      // Durations of jobs

XPRBindexSet Jobs;      // Names of Jobs
XPRBvar *start;         // Start times of jobs
XPRBvar **delta;        // Binaries for start times
XPRBvar z;              // Max. completion time

XPRBprob p("Jobs");     // Initialize BCL & a new problem

void readData()
{
    char name[100];
    FILE *datafile;

    // Create a new index set
    Jobs = p.newIndexSet("jobs", MAXNJ);
}

```

```

// Open data file for read access
datafile=fopen("durations.dat","r");
// Read in all (non-empty) lines up to the end of the file
while(NJ<MAXNJ &&
      XPRBreadlinecb(XPRB_FGETS, datafile, 99, "T,d", name, &DUR[NJ]))
{
    Jobs += name;          // Add job to the index set
    NJ++;
}
fclose(datafile);          // Close the input file
cout << "Number of jobs read: " << Jobs.getSize() << endl;
}

void jobsModel()
{
    XPRBexpr le;
    int j,t;

    // Create start time variables with bounds
    start = new XPRBvar[NJ];
    if(start==NULL)
    { cout << "Not enough memory for 'start' variables." << endl;
      exit(0); }
    for(j=0;j<NJ;j++)
        start[j] = p.newVar("start",XPRB_PL,0,NT-DUR[j]+1));
    z = p.newVar("z",XPRB_PL,0,NT); // Makespan variable

    delta = new XPRBvar*[NJ];
    if(delta==NULL)
    { cout << "Not enough memory for 'delta' variables." << endl;
      exit(0); }
    for(j=0;j<NJ;j++) // Binaries for each job
    {
        delta[j] = new XPRBvar[NT];
        if(delta[j]==NULL)
        { cout <<"Not enough memory for 'delta_j' variables." << endl;
          exit(0); }
        for(t=0;t<(NT-DUR[j]+1);t++)
            delta[j][t] =
                p.newVar(XPRBnewname("delta%s_%d",Jobs[j],t+1), XPRB_BV);

        for(j=0;j<NJ;j++) // Calculate max. completion time
            p.newCtr("Makespan", start[j]+DUR[j] <= z);
        // Precedence relation betw. jobs
        p.newCtr("Prec", start[0]+DUR[0] <= start[2]);

        for(j=0;j<NJ;j++) // Linking start times & binaries
        {
            le=0;
            for(t=0;t<(NT-DUR[j]+1);t++) le += (t+1)*delta[j][t];
            p.newCtr(XPRBnewname("Link_%d",j+1), le == start[j]);
        }

        for(j=0;j<NJ;j++) // Unique start time for each job
        {
            le=0;
            for(t=0;t<(NT-DUR[j]+1);t++) le += delta[j][t];
            p.newCtr(XPRBnewname("One_%d",j+1), le == 1);
        }

        p.setObj(z); // Define and set objective
        jobsSolve(); // Solve the problem

        delete [] start;
        for(j=0;j<NJ;j++) delete [] delta[j];
        delete [] delta;
    }

int main(int argc, char **argv)
{
    readData(); // Read in the data
    jobsModel(); // Problem definition
    return 0;
}

```

5.1.2 QCQP Example

The following is an implementation with BCL C++ of the QCQP example described in Section 3.4.1:

```
#include <iostream>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

#define N 42
double CX[N], CY[N], R[N];

... // Initialize the data arrays

int main(int argc, char **argv)
{
    int i, j;
    XPRBvar x[N], y[N];
    XPRBexpr qe;
    XPRBctr cobj, c;
    XPRBprob prob("airport"); // Initialize a new problem in BCL

    /**** VARIABLES ****/
    for(i=0; i<N; i++)
        x[i] = prob.newVar(XPRBnewname("x(%d)", i+1), XPRB_PL, -10, 10);
    for(i=0; i<N; i++)
        y[i] = prob.newVar(XPRBnewname("y(%d)", i+1), XPRB_PL, -10, 10);

    /****OBJECTIVE****/
    // Minimize the total distance between all points
    qe=0;
    for(i=0; i<N-1; i++)
        for(j=i+1; j<N; j++) qe+= sqrt(x[i]-x[j])+sqrt(y[i]-y[j]);
    cobj = prob.newCtr("TotDist", qe);
    prob.setObj(cobj); // Set objective function

    /**** CONSTRAINTS ****/
    // All points within given distance of their target location
    for(i=0; i<N; i++)
        c = prob.newCtr("LimDist", sqrt(x[i]-CX[i])+sqrt(y[i]-CY[i]) <= R[i]);

    /****SOLVING + OUTPUT****/
    prob.setSense(XPRB_MINIM); // Choose sense of optimization
    prob.solve(""); // Solve the problem

    cout << "Solution: " << prob.getObjVal() << endl;
    for(i=0; i<N; i++)
    {
        cout << x[i].getName() << ": " << x[i].getSol() << ", ";
        cout << y[i].getName() << ": " << y[i].getSol() << endl;
    }

    return 0;
}
```

5.2 C++ class reference

The complete set of classes of the BCL C++ interface is summarized in the following list:

XPRB	Initialization and general settings.	p. 165
XPRBbasis	Methods for accessing bases.	p. 168
XPRBctr	Methods for modifying and accessing constraints and operators for constructing them.	p. 170

<code>XPRBcut</code>	Methods for modifying and accessing cuts and operators for constructing them. p. 180
<code>XPRBexpr</code>	Methods and operators for constructing linear and quadratic expressions. p. 186
<code>XPRBindexSet</code>	Methods for accessing index sets and operators for adding and retrieving set elements. p. 191
<code>XPRBprob</code>	Problem definition, including methods for creating and deleting the modeling objects, problem solving, changing settings, and retrieving solution information. p. 195
<code>XPRBrelation</code>	Methods and operators for constructing linear or quadratic relations from expressions. p. 216
<code>XPRBsos</code>	Methods for modifying and accessing Special Ordered Sets and operators for constructing them. p. 217
<code>XPRBvar</code>	Methods for modifying and accessing variables. p. 221

The method `isValid` may require some explanation: it should be used in combination with methods `getVarByName`, `getCtrByName` etc. These methods always return an object of the desired type, unlike the corresponding functions in standard BCL which return a `NULL` pointer if the object was not found. Only with method `isValid` it is possible to test whether the object is a valid object, that is, whether it is contained in a problem definition.

All C++ classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsos`, `XPRBindexSet`, `XPRBbasis`) take the same names, with the exception of `XPRBindexSet`. The corresponding BCL modeling object in C can be obtained from each of these classes, with the method `getCRef`. It is also possible to obtain the Xpress-Optimizer problem corresponding to a BCL C++ problem by using method `XPRBprob.getXPRSProb`. Please see Section B.6 for further detail on using BCL C++ with the Optimizer library.

Most of the methods of the classes with direct correspondence to C modeling objects call standard BCL C functions, as indicated, and return their result.

The major difference between the C and C++ interfaces is in the way linear and quadratic expressions and constraints are created. In C++, the algebraic operators like `+` or `==` are overloaded so that constraints may be written in a form that is close to an algebraic formulation.

Some additional classes have been introduced to aid the termwise definition of constraints with overloaded arithmetic operators. Linear and quadratic expressions (class `XPRBexpr`) are required in the definition of constraints and Special Ordered Sets. Linear and quadratic relations (class `XPRBrelation`), may be used as an intermediary in the definition of constraints.

Another class that does not correspond to any standard BCL modeling object is the class `XPRB` that contains methods relating to the initialization of BCL and the general status of the software.

XPRB

Description

Initialization and general settings.

Methods

```
int getTime();  
    Get the running time.  
const char *getVersion();  
    Get the version number of BCL.  
int init();  
    Initialize BCL.  
int setColOrder(int num);  
    Set a column ordering criterion for matrix generation.  
int setMsgLevel(int lev);  
    Set the message print level.  
int setRealFmt(String fmt);  
    Set the format for printing real numbers.
```

Method detail

getTime

Synopsis `int getTime();`

Return value System time measure in milliseconds.

Description This methods returns the system time measure in milliseconds. The absolute value is system-dependent. To measure the execution time of a program, this methods can be used to calculate the difference between the start time and the time at the desired point in the program.

Example This example shows how to measure the elapsed time in a BCL program:

```
int starttime;  
XPRB::init();  
starttime = XPRB::getTime();  
...  
cout << "Time: " << (XPRB::getTime()-starttime)/1000;  
cout << " sec" << endl;
```

Related topics Calls `XPRBgetttime`

getVersion

Synopsis `const char *getVersion();`

Return value BCL version number if function executed successfully, `NULL` otherwise.

Description The version number returned by this method is required if the user is reporting a problem.

Example The following example retrieves and prints out the BCL version number:

```

const char *version;
XPRB::init();
version = XPRB::getVersion();
cout << "Xpress-BCL version " << version << endl;

```

Related topics Calls [XPRBgetversion](#)

init

Synopsis `int init();`

Return value 0 if initialization executed successfully, 1 otherwise.

Description This method explicitly initializes BCL, that is it tests whether a license for running this software is available. Without this explicit initialization the initialization will be performed at the creation of the first problem (see [XPRBprob](#)). There is no need to call this explicit initialization unless you wish to separate the license check from problem creation or perform some general settings before creating any problem. This method also initializes Xpress-Optimizer.

Example This example shows how to initialize BCL explicitly before creating a problem.

```

XPRBprob *prob;

if (XPRB::init())
{ cout << "Initialization failed" << endl; return 1; }
prob = new XPRBprob("myprob");

```

Related topics Calls [XPRBinit](#)

setColOrder

Synopsis `int setColOrder(int num);`

Argument `num` The ordering flag, which must be one of:
 0 default ordering;
 1 alphabetical order.

Return value 0 if method executed successfully, 1 otherwise.

Description 1. BCL runs reproduce always the same matrix for a problem. This method allows the user to choose a different ordering criterion than the default one. Note that this method only changes the order of columns in what is sent to Xpress-Optimizer, you do not see any effect when exporting the matrix with BCL. However you can control the effect by exporting the matrix from the Optimizer.

2. The setting applies to all problems that are created subsequently. It is also possible to change the setting for a particular problem (see [XPRBprob](#)).

Related topics Calls [XPRBsetcolorder](#)

setMsgLevel

Synopsis `int setMsgLevel(int lev);`

Argument	level The message level, <i>i.e.</i> the type of messages printed by BCL. This may be one of: <ul style="list-style-type: none"> 0 no messages printed; 1 error messages only printed; 2 warnings and errors printed; 3 warnings, errors, and Optimizer log printed (default); 4 all messages printed.
Return value	0 if method executed successfully, 1 otherwise.
Description	<ol style="list-style-type: none"> 1. BCL can produce different types of messages; status information, warnings and errors. This function controls which of these are output. For settings 1 or higher, the corresponding Optimizer output is also displayed. In addition to this setting, the amount of Optimizer output can be modified through several Optimizer printing control parameters (see the 'Xpress-Optimizer Reference Manual'). 2. The setting applies to all problems that are created subsequently. It is also possible to change the setting for a particular problem (see XPRBprob).
Example	See XPRBprob.setMsgLevel .
Related topics	Calls XPRBsetmsglevel

setRealFmt

Synopsis	<code>int setRealFmt(String fmt);</code>
Argument	fmt Format string (as used by the C function <code>printf</code>). Simple format strings are of the form <code>%n</code> where <code>n</code> may be, for instance, one of <ul style="list-style-type: none"> <code>g</code> default printing format (precision: 6 digits; exponential notation if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision) <code>.numf</code> print real numbers in the style <code>[-]d.d</code> where the number of digits after the decimal point is equal to the given precision <code>num</code>.
Return value	0 if method executed successfully, 1 otherwise.
Description	<ol style="list-style-type: none"> 1. In problems with very large or very small numbers it may become necessary to change the printing format to obtain a more exact output. In particular, by changing the precision it is possible to reduce the difference between matrices loaded in memory into Xpress-Optimizer and the output produced by exporting them to a file. 2. The setting applies to all problems that are created subsequently. It is also possible to change the setting for a particular problem (see XPRBprob).
Example	<p>This example sets the BCL number printing format to 8 digits after the decimal point. It then creates a problem and changes the number printing format for this problem back to the default:</p> <pre> XPRBprob *prob; XPRB::init(); XPRB::setRealFmt("%.10f"); prob = new XPRBprob("myprob"); prob->setRealFmt("%g"); </pre>
Related topics	Calls XPRBsetrealfmt

XPRBbasis

Description

Methods for accessing bases.

Constructors

```
XPRBbasis();  
XPRBbasis(xbbasis *bs);
```

Methods

```
xbbasis *getCRef();  
    Get the C modeling object.  
bool isValid();  
    Test the validity of the basis object.  
void reset();  
    Reset the basis object.
```

Constructor detail

XPRBbasis

Synopsis	<pre>XPRBbasis(); XPRBbasis(xbbasis *bs);</pre>
Argument	<pre>bs</pre> A basis in BCL C.
Description	Create a new basis object.

Method detail

getCRef

Synopsis	<pre>xbbasis *getCRef();</pre>
Return value	The underlying modeling object in BCL C.
Description	This method returns the basis object in BCL C that belongs to the C++ basis object.

isValid

Synopsis	<pre>bool isValid();</pre>
Return value	<code>true</code> if object is valid, <code>false</code> otherwise.
Description	This method checks whether the basis object is correctly defined.

reset

Synopsis	<pre>void reset();</pre>
Description	Clear the definition of the basis object; includes deletion of the underlying C object.

Example See `XPRBprob.saveBasis`.

Related topics Calls `XPRBdelbasis`

XPRBctr

Description

Methods for modifying and accessing constraints and operators for constructing them.

Constructors

```
XPRBctr();  
XPRBctr(xbctr *c);  
XPRBctr(xbctr *c, XPRBrelation& r);
```

Methods

```
void add(XPRBexpr& e);  
    Add an expression to a constraint.  
int addTerm(XPRBvar& var, double val);  
int addTerm(double val, XPRBvar& var);  
int addTerm(XPRBvar& var);  
int addTerm(double val);  
int addTerm(XPRBvar& var, XPRBvar& var2, double val);  
int addTerm(double val, XPRBvar& var, XPRBvar& var2);  
int addTerm(XPRBvar& var, XPRBvar& var2);  
    Add a term to a constraint.  
int delTerm(XPRBvar& var);  
int delTerm(XPRBvar& var, XPRBvar& var2);  
    Delete a term from a constraint.  
double getAct();  
    Get activity value for a constraint.  
xbctr *getCRef();  
    Get the C modeling object.  
double getDual();  
    Get dual value.  
const char *getName();  
    Get the name of a constraint.  
int getRange(double *lw, double *up);  
    Get the range values for a range constraint.  
double getRangeL();  
    Get the lower range bound for a range constraint.  
double getRangeU();  
    Get the upper range bound for a range constraint.  
double getRHS();  
    Get the right hand side value of a constraint.  
double getRNG(int rngtype);  
    Get ranging information for a constraint.  
int getRowNum();  
    Get the row number for a constraint.  
double getSlack();  
    Get slack value for a constraint.  
int getType();  
    Get the row type of a constraint.  
bool isModCut();  
    Check the type of a constraint.  
bool isValid();  
    Test the validity of the constraint object.
```

```

int print();
    Print out a constraint.
void reset();
    Reset the constraint object.
int setModCut(bool mstat);
    Set the constraint type.
int setRange(double lw, double up);
    Define a range constraint.
int setTerm(XPRBvar& var, double val);
int setTerm(double val, XPRBvar& var);
int setTerm(double val);
int setTerm(XPRBvar& var, XPRBvar& var2, double val);
int setTerm(double val, XPRBvar& var, XPRBvar& var2);
    Set a constraint term.
int setType(int type);
    Set the constraint type.

```

Operators

Assigning constraints and adding (linear or quadratic) expressions:

```

ctr = rel
ctr += expr
ctr -= expr

```

Constructor detail

XPRBctr

Synopsis	<pre> XPRBctr(); XPRBctr(xbctr *c); XPRBctr(xbctr *c, XPRBrelation& r); </pre>
Arguments	<p>c A constraint in BCL C.</p> <p>r Relation defining the constraint.</p>
Description	Create a new constraint object.

Method detail

add

Synopsis	<code>void add(XPRBexpr& e);</code>
Argument	e A linear or quadratic expression (may be just a single variable or a constant).
Description	This method adds a linear or quadratic expression to the left hand side of a constraint. That means, if the expression contains a constant, this value is subtracted from the constant representing the right hand side of the constraint.
Example	See <code>XPRBctr.setTerm</code> .

addTerm

Synopsis

```
int addTerm(XPRBvar& var, double val);
int addTerm(double val, XPRBvar& var);
int addTerm(XPRBvar& var);
int addTerm(double val);
int addTerm(XPRBvar& var, XPRBvar& var2, double val);
int addTerm(double val, XPRBvar& var, XPRBvar& var2);
int addTerm(XPRBvar& var, XPRBvar& var2);
```

Arguments

var A BCL variable.

var2 A second BCL variable (may be the same as **var**).

val Value of the coefficient of the variable **var**.

Return value

0 if method executed successfully, 1 otherwise.

Description

This method adds a new term to a constraint, comprising the variable **var** (or the product of variables **var** and **var2**) with coefficient **val**. If the constraint already has a term with variable **var** (respectively variables **var** and **var2**), **val** is added to its coefficient. If no variable is specified, the value **val** is added to the right hand side of the constraint. Constraint terms can also be added with method [XPRBctr.add](#).

Example

See [XPRBctr.setTerm](#).

Related topics

Calls [XPRBaddterm](#)

delTerm

Synopsis

```
int delTerm(XPRBvar& var);
int delTerm(XPRBvar& var, XPRBvar& var2);
```

Arguments

var A BCL variable.

var2 A second BCL variable (may be the same as **var**).

Return value

0 if method executed successfully, 1 otherwise.

Description

This function deletes a variable term from the given constraint. The constant term (right hand side value) is changed/reset with method [XPRBctr.setTerm](#).

Related topics

Calls [XPRBdelterm](#)

getAct

Synopsis

```
double getAct();
```

Return value

Activity value for the constraint, 0 in case of an error.

Description

This method returns the activity value for a constraint. It may be used with constraints that are not part of the problem (in particular, constraints without relational operators, that is, constraints of type `XPRB_N`). In this case the function returns the evaluation of the constraint terms involving variables that are in the problem. Otherwise, the constraint activity is calculated as $activity = RHS - slack$. If this method is called after completion of a global search and an integer solution has

been found (that is, if method `XPRBprob.getMIPStat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value corresponding to the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the activity value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBprob.sync` with the flag `XPRB_XPRS_SOL`.

Example

The following example shows how to retrieve solution values and some other information for a constraint.

```
XPRBvar x,y;
XPRBctr Ctrl;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);
y = prob.newVar("y", XPRB_PL, 0, 200);
Ctrl = prob.newCtr("C1", 3*x + 2*y <= 400);

... // Solve an LP problem

if (Ctrl.getRowNum() >= 0 && prob.getLPStat()==XPRB_LP_OPTIMAL)
{
    cout << Ctrl.getName() << ": activity: " << Ctrl.getAct();
    cout << " = " << Ctrl.getRHS() << " - " << Ctrl.getSlack();
    cout << ", dual: " << Ctrl.getDual() << endl;
}
else
    cout << "No solution information available." << endl;
```

Related topics

Calls `XPRBgetact`

getCRef

Synopsis

```
xbctr *getCRef();
```

Return value

The underlying modeling object in BCL C.

Description

This method returns the constraint object in BCL C that belongs to the C++ constraint object.

getDual

Synopsis

```
double getDual();
```

Return value

Dual value for the constraint, 0 in case of an error.

Description

This function returns the dual value for a constraint. The user may wish to test first whether this constraint is part of the problem, for instance by checking that the row number is non-negative.

If this function is called after completion of a global search and an integer solution has been found (that is, if function `XPRBprob.getMIPStat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the dual value in the last LP that has been

solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBprob.sync` with the flag `XPRB_XPRS_SOL`.

Example See `XPRBctr.getAct`.

Related topics Calls `XPRBgetdual`

getName

Synopsis `const char *getName();`

Return value Name of the constraint if function executed successfully, `NULL` otherwise

Description This method returns the name of a constraint. If the user has not defined a name the default name generated by BCL is returned.

Example See `XPRBctr.getAct`.

Related topics Calls `XPRBgetctrname`

getRange

Synopsis `int getRange(double *lw, double *up);`

Arguments
`lw` Lower bound on the range constraint.
`up` Upper bound on the range constraint.

Return value 0 if method executed successfully, 1 otherwise.

Description This method returns the range values of the given constraint.

Related topics Calls `XPRBgetrange`

getRangeL

Synopsis `double getRangeL();`

Return value Lower bound on the range constraint.

Description This method returns the lower bound on the range defined for the given constraint.

Example See `XPRBctr.setRange`.

Related topics Calls `XPRBgetrange`

getRangeU

Synopsis `double getRangeU();`

Return value Upper bound on the range constraint.

Description	This method returns the upper bound on the range defined for the given constraint.
Example	See <code>XPRBctr.setRange</code> .
Related topics	Calls <code>XPRBgetrange</code>

getRHS

Synopsis	<code>double getRHS();</code>
Return value	Right hand side value of the constraint, 0 in case of an error.
Description	This method returns the right hand side value (<i>i.e.</i> the constant term) of a previously defined constraint. The default right hand side value is 0. If the given constraint is a ranged constraint this function returns its upper bound.
Example	See <code>XPRBctr.getAct</code> .
Related topics	Calls <code>XPRBgetrhs</code>

getRNG

Synopsis	<code>double getRNG(int rngtype);</code>								
Argument	<p><code>rngtype</code> The type of ranging information sought. This is one of:</p> <table> <tr> <td><code>XPRB_UPACT</code></td><td>upper activity;</td></tr> <tr> <td><code>XPRB_LOACT</code></td><td>lower activity;</td></tr> <tr> <td><code>XPRB_UUP</code></td><td>upper unit cost;</td></tr> <tr> <td><code>XPRB_UDN</code></td><td>lower unit cost.</td></tr> </table>	<code>XPRB_UPACT</code>	upper activity;	<code>XPRB_LOACT</code>	lower activity;	<code>XPRB_UUP</code>	upper unit cost;	<code>XPRB_UDN</code>	lower unit cost.
<code>XPRB_UPACT</code>	upper activity;								
<code>XPRB_LOACT</code>	lower activity;								
<code>XPRB_UUP</code>	upper unit cost;								
<code>XPRB_UDN</code>	lower unit cost.								
Return value	Ranging information of the required type.								
Description	This method can only be used after solving an LP problem. Ranging information for MIP problems can be obtained by fixing all discrete variables to their solution values and re-solving the resulting LP problem.								
Example	<p>The following example displays the constraint activity and the activity range.</p> <pre> XPRBvar x,y; XPRBctr Ctrl; XPRBprob prob("myprob"); x = prob.newVar("x", XPRB_PL, 0, 200); y = prob.newVar("y", XPRB_PL, 0, 200); Ctrl = prob.newCtr("C1", 3*x + 2*y <= 400); ... // Solve the problem cout << "C1: " << Ctrl.getAct() << " (activity range: "; cout << Ctrl.getRNG(XPRB_LOACT) << ", " ; cout << Ctrl.getRNG(XPRB_UPACT) << ")" << endl; </pre>								
Related topics	Calls <code>XPRBgetctrrng</code>								

getRowNum

Synopsis	<code>int getRowNum();</code>
Return value	Row number (non-negative value), or a negative value.
Description	This method returns the matrix row number of a constraint. If the matrix has not yet been generated or the constraint is not part of the matrix (constraint type <code>XPRB_N</code> or no non-zero terms) then the return value is negative. To check whether the matrix has been generated, use method <code>XPRBprob.getProbStat</code> . The counting of row numbers starts with 0.
Example	See <code>XPRBctr.getAct</code> .
Related topics	Calls <code>XPRBgetrownum</code>

getSlack

Synopsis	<code>double getSlack();</code>
Return value	Slack value for the constraint, 0 in case of an error.
Description	<p>This method returns the slack value for a constraint. The user may wish to test first whether this constraint is part of the problem, for instance by checking that the row number is non-negative.</p> <p>If this function is called after completion of a global search and an integer solution has been found (that is, if method <code>XPRBprob.getMIPStat</code> returns values <code>XPRB_MIP_SOLUTION</code> or <code>XPRB_MIP_OPTIMAL</code>), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the slack value in the last LP that has been solved. If this function is used <i>during</i> the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to <code>XPRBprob.sync</code> with the flag <code>XPRB_XPRS_SOL</code>.</p>
Example	See <code>XPRBctr.getAct</code> .
Related topics	Calls <code>XPRBgetslack</code>

getType

Synopsis	<code>int getType();</code>
Return value	<code>XPRB_L</code> 'less than or equal to' inequality; <code>XPRB_G</code> 'greater than or equal to' inequality; <code>XPRB_E</code> equality; <code>XPRB_N</code> a non-binding row (objective function); <code>XPRB_R</code> a range constraint; <code>-1</code> an error has occurred.
Description	This method returns the constraint type if successful, and -1 in case of an error.
Example	See <code>XPRBctr.setRange</code> .
Related topics	Calls <code>XPRBgetctrtype</code>

isModCut

Synopsis	<code>bool isModCut();</code>
Return value	<code>true</code> if constraint is a model cut, <code>false</code> otherwise.
Description	This method indicates whether the given constraint is a model cut or an ordinary constraint.
Related topics	Calls <code>XPRBgetmodcut</code>

isValid

Synopsis	<code>bool isValid();</code>
Return value	<code>true</code> if object is valid, <code>false</code> otherwise.
Description	This method checks whether the constraint object is correctly defined. It should always be used to test the result returned by <code>XPRBprob.getCtrByName</code> .
Example	See <code>XPRBprob.getCtrByName</code> .

print

Synopsis	<code>int print();</code>
Return value	0 if function executed successfully, 1 otherwise.
Description	This method prints out a constraint in LP format. It is not available in the student version.
Example	See <code>XPRBctr.setRange</code> .
Related topics	Calls <code>XPRBprintctr</code>

reset

Synopsis	<code>void reset();</code>
Description	Clear the definition of the constraint object.

setModCut

Synopsis	<code>int setModCut(bool mstat);</code>						
Argument	<table><tr><td><code>mstat</code></td><td>The constraint type, which must be one of:</td></tr><tr><td><code>false</code></td><td>constraint;</td></tr><tr><td><code>true</code></td><td>model cut.</td></tr></table>	<code>mstat</code>	The constraint type, which must be one of:	<code>false</code>	constraint;	<code>true</code>	model cut.
<code>mstat</code>	The constraint type, which must be one of:						
<code>false</code>	constraint;						
<code>true</code>	model cut.						

Return value	0 if method executed successfully, 1 otherwise.
Description	This method changes the type of a previously defined constraint from ordinary constraint to model cut and vice versa.
Example	<p>The following example turns the constraint <code>Ctr3</code> into a model cut.</p> <pre> XPRBvar y,b; XPRBctr Ctr3; XPRBprob prob("myprob"); y = prob.newVar("y", XPRB_PL, 0, 200); b = prob.newVar("b", XPRB_BV); Ctr3 = prob.newCtr("C3", y >= 50*b); Ctr3.setModCut(true); </pre>
Related topics	Calls XPRBsetmodcut

setRange

Synopsis	<code>int setRange(double lw, double up);</code>
Arguments	<p><code>lw</code> Lower bound on the range constraint.</p> <p><code>up</code> Upper bound on the range constraint.</p>
Return value	0 if method executed successfully, 1 otherwise.
Description	This method changes the type of a constraint to a range constraint within the bounds specified by <code>lw</code> and <code>up</code> . The constraint type and right hand side value of the constraint are replaced by the type <code>XPRB_R</code> (range) and the two bounds.
Example	<p>The following example defines a constraint with the range bounds 100 and 500, adds 5 to the range bounds and prints them out. The constraint is then changed to an inequality constraint whereby the upper range bound is transformed into the right hand side. The output printed by this example is displayed in the commentaries.</p> <pre> XPRBvar x,y; XPRBctr Ctrl; XPRBprob prob("myprob"); x = prob.newVar("x", XPRB_PL, 0, 200); y = prob.newVar("y", XPRB_PL, 0, 200); Ctrl = prob.newCtr("C1", 3*x + 2*y <= 400); Ctrl.setRange(100,500); Ctrl.addTerm(5); if (Ctrl.getType() == XPRB_R) { cout << "C1 in [" << Ctrl.getRangeL() << ", "; cout << Ctrl.getRangeU() << "]" << endl; // C1 in [105,505] Ctrl.setType(XPRB_G); Ctrl.print(); // C1: 3*x + 2*y >= 505 } </pre>
Related topics	Calls XPRBsetrange

setTerm

Synopsis

```
int setTerm(XPRBvar& var, double val);
int setTerm(double val, XPRBvar& var);
int setTerm(double val);
int setTerm(XPRBvar& var, XPRBvar& var2, double val);
int setTerm(double val, XPRBvar& var, XPRBvar& var2);
```

Arguments

var A BCL variable.

var2 A second BCL variable (may be the same as **var**).

val Value of the coefficient of the variable **var**.

Return value

0 if method executed successfully, 1 otherwise.

Description

This method sets the coefficient of a variable (or of the product of the two given variables) to the value **val**. If no variable is specified, the right hand side of the constraint is set to **val**.

Example

This example sets the coefficient of variable **y** in constraint **Ctrl** to 5 and then adds a linear expression and a constant term. The commentaries show the constraint definitions resulting from the modifications. Please notice in particular the different behavior of **add** and **addTerm** for the addition of constants.

```
XPRBvar x,y;
XPRBctr Ctrl;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);
y = prob.newVar("y", XPRB_PL, 0, 200);

Ctrl = prob.newCtr("C1", 3*x + 2*y <= 400);
Ctrl.setTerm(5, y);           // C1: 3*x + 5*y <= 400
Ctrl.add(x+10);               // C1: 4*x + 5*y <= 390
Ctrl.setTerm(400);            // C1: 4*x + 5*y <= 400
Ctrl.addTerm(5);               // C1: 4*x + 5*y <= 405
```

Related topics

Calls [XPRBsetterm](#)

setType

Synopsis

```
int setType(int type);
```

Argument

type The constraint type, which must be one of:

- XPRB_L** 'less than or equal to' constraint;
- XPRB_G** 'greater than or equal to' constraint;
- XPRB_E** an equality;
- XPRB_N** a non-binding row (objective function).

Return value

0 if method executed successfully, 1 otherwise.

Description

This method changes the type of a previously defined constraint to inequality, equation or non-binding. Method [XPRBctr.setRange](#) has to be used for changing the constraint to a ranged constraint. If a ranged constraint is changed back to some other type with this method, its upper bound becomes the right hand side value.

Example

See [XPRBctr.setRange](#).

Related topics

Calls [XPRBsetctrtype](#)

XPRBCut

Description

Methods for modifying and accessing cuts and operators for constructing them.

Constructors

```
XPRBCut();  
XPRBCut(xbcut *c);  
XPRBCut(xbcut *c, XPRBrelation& r);
```

Methods

```
void add(XPRBexpr& le);  
    Add a linear expression to a cut.  
int addTerm(XPRBvar& var, double val);  
int addTerm(double val, XPRBvar& var);  
int addTerm(XPRBvar& var);  
int addTerm(double val);  
    Add a term to a cut.  
int delTerm(XPRBvar& var);  
    Delete a term from a cut.  
xbcut *getCRef();  
    Get the C modeling object.  
int getID();  
    Get the classification or identification number of a cut.  
double getRHS();  
    Get the RHS value of a cut.  
int getType();  
    Get the type of a cut.  
bool isValid();  
    Test the validity of the cut object.  
int print();  
    Print out a cut.  
void reset();  
    Reset the cut object.  
int setID(int id);  
    Set the classification or identification number of a cut.  
int setTerm(XPRBvar& var, double val);  
int setTerm(double val, XPRBvar& var);  
int setTerm(XPRBvar& var);  
int setTerm(double val);  
    Set a cut term.  
int setType(int type);  
    Set the type of a cut.
```

Operators

Assigning cuts and adding linear expressions:

```
cut = linrel  
cut += linexp  
cut -= linexp
```

Constructor detail

XPRBcut

Synopsis	<pre>XPRBcut(); XPRBcut(xbcut *c); XPRBcut(xbcut *c, XPRBrelation& r);</pre>
Arguments	<p>c A cut in BCL C.</p> <p>r Linear relation defining the cut.</p>
Description	Create a new cut object.

Method detail

add

Synopsis	<pre>void add(XPRBexpr& le);</pre>
Argument	le A linear expression (may be a single variable or a constant).
Return value	0 if method executed successfully, 1 otherwise.
Description	This method adds a linear expression to a cut. That means, if the linear expression contains a constant, this value is subtracted from the constant representing the right hand side of the cut.
Example	This example defines a cut and then modifies its definition by adding a terms and changing the coefficient of a variable. The resulting cut definitions (as displayed by <code>XPRBcut.print</code>) are shown as comments. Please notice in particular the different behavior of <code>add</code> and <code>addTerm</code> for the addition of constants.

```
XPRBvar x,y,b;  
XPRBcut Cut2;  
XPRBprob prob("myprob");  
  
x = prob.newVar("y", XPRB_PL, 0, 200);  
y = prob.newVar("y", XPRB_PL, 0, 200);  
b = prob.newVar("b", XPRB_BV);  
  
Cut2 = prob.newCut(y <= 100*b, 1);  
Cut2.add(x+2);           // x + y - 100*b <= -2  
Cut2.delTerm(x);         //      y - 100*b <= -2  
Cut2.setTerm(0);        //      y - 100*b <= 0  
Cut2 += x+2;            // x + y - 100*b <= -2  
Cut2.addTerm(2);        // x + y - 100*b <= 0  
Cut2.setTerm(y, -5);    // x - 5*y - 100*b <= 0
```

addTerm

Synopsis	<pre>int addTerm(XPRBvar& var, double val); int addTerm(double val, XPRBvar& var); int addTerm(XPRBvar& var); int addTerm(double val);</pre>
Arguments	<p>var A BCL variable.</p> <p>val Value of the coefficient of the variable var.</p>

Return value	0 if method executed successfully, 1 otherwise.
Description	This method adds a new term to a cut, comprising the variable <code>var</code> with coefficient <code>val</code> . If the cut already has a term with variable <code>var</code> , <code>val</code> is added to its coefficient. If no variable is specified, the value <code>val</code> is added to the right hand side of the cut. Cut terms can also be added with method <code>XPRBcut.add</code> .
Example	See <code>XPRBcut.add</code> .
Related topics	Calls <code>XPRBaddcutterm</code>

delTerm

Synopsis	<code>int delTerm(XPRBvar& var);</code>
Argument	<code>var</code> A BCL variable.
Return value	0 if method executed successfully, 1 otherwise.
Description	This method removes a variable term from a cut. The constant term (right hand side value) is changed/reset with method <code>XPRBcut.setTerm</code> .
Example	See <code>XPRBcut.add</code> .
Related topics	Calls <code>XPRBdelcutterm</code>

getCRef

Synopsis	<code>xbcut *getCRef();</code>
Return value	The underlying modeling object in BCL C.
Description	This method returns the cut object in BCL C that belongs to the C++ cut object.

getID

Synopsis	<code>int getID();</code>
Return value	Classification or identification number.
Description	This method returns the classification or identification number of a cut.
Example	See <code>XPRBcut.setID</code> .
Related topics	Calls <code>XPRBgetcutid</code>

getRHS

Synopsis	<code>double getRHS();</code>
Return value	Right hand side (RHS) value (default 0).

Description	This method returns the RHS value (= constant term) of a previously defined cut. The default RHS value is 0.
Related topics	Calls XPRBgetcutrhs

getType

Synopsis	<code>int getType();</code>
Return value	<code>XPRB_L</code> \leq (inequality) <code>XPRB_G</code> \geq (inequality) <code>XPRB_E</code> = (equation) <code>-1</code> An error has occurred,

Description This method returns the type of the given cut.

Related topics Calls [XPRBgetcuttype](#)

isValid

Synopsis	<code>bool isValid();</code>
Return value	<code>true</code> if object is valid, <code>false</code> otherwise.
Description	This method checks whether the cut object is correctly defined.

print

Synopsis	<code>int print();</code>
Return value	0 if function executed successfully, 1 otherwise.
Description	This function prints out a cut in LP format. It is not available in the student version.
Example	See XPRBCut.setID .
Related topics	Calls XPRBprintcut

reset

Synopsis	<code>void reset();</code>
Description	Clear the definition of the cut object.

setID

Synopsis	<code>int setID(int id);</code>
Argument	<code>id</code> Classification or identification number.
Return value	0 if method executed successfully, 1 otherwise.
Description	This function changes the classification or identification number of a previously defined cut. This change does not have any effect on the cut definition in Xpress-Optimizer if the cut has already been added to the matrix with <code>XPRBprob.addCuts</code> .
Example	<p>This example defines a cut and then modifies its ID and relation type. The resulting output is shown in the comment.</p> <pre>XPRBvar y,b; XPRBcut Cut1; XPRBprob prob("myprob"); y = prob.newVar("y", XPRB_PL, 0, 200); b = prob.newVar("b", XPRB_BV); Cut1 = prob.newCut(y == 100*b); Cut1.setID(1); if (Cut1.getID()>0) Cut1.setType(XPRB_G); Cut1.print(); // CUT(1): y - 100*b >= 0</pre>
Related topics	Calls <code>XPRBsetcutid</code>

setTerm

Synopsis	<pre>int setTerm(XPRBvar& var, double val); int setTerm(double val, XPRBvar& var); int setTerm(XPRBvar& var); int setTerm(double val);</pre>
Arguments	<code>var</code> A BCL variable. <code>val</code> Value of the coefficient of the variable <code>var</code> .
Return value	0 if method executed successfully, 1 otherwise.
Description	This function sets the coefficient of a variable to the value <code>val</code> . If no variable is specified, the right hand side of the cut is set to <code>val</code> .
Example	See <code>XPRBcut.add</code> .
Related topics	Calls <code>XPRBsetcutterm</code>

setType

Synopsis	<code>int setType(int type);</code>
-----------------	-------------------------------------

Argument	<div> <div>type</div> <div>Type of the cut:</div> <div> XPRB_L \leq (inequality) XPRB_G \geq (inequality) XPRB_E $=$ (equation) </div> </div>
Return value	0 if method executed successfully, 1 otherwise.
Description	This function changes the type of the given cut. This change does not have any effect on the cut definition in Xpress-Optimizer if the cut has already been added to the matrix with the method <code>XPRBprob.addCuts</code> .
Example	See <code>XPRBcut.setID</code> .
Related topics	Calls <code>XPRBsetcuttype</code>

XPRBExpr

Description

Methods and operators for constructing linear and quadratic expressions.

Constructors

```
XPRBExpr(double d);
XPRBExpr(int i);
XPRBExpr(double d, XPRBvar& v);
XPRBExpr(double d, XPRBvar& v, XPRBvar& v2);
XPRBExpr(XPRBvar& v);
XPRBExpr(XPRBExpr& e);
```

Methods

```
XPRBExpr& add(XPRBExpr& e);
XPRBExpr& add(XPRBvar& v);
    Addition to an expression
XPRBExpr& assign(XPRBExpr& e);
    Copy an expression.
int delTerm(XPRBvar& var);
int delTerm(XPRBvar& var, XPRBvar& var2);
    Delete a term from an expression.
double getSol();
    Get evaluation of an expression.
XPRBExpr& mul(double d);
XPRBExpr& mul(XPRBExpr& e);
    Multiply an expression by a constant factor or an expression.
XPRBExpr& neg();
    Negation of an expression.
int setTerm(XPRBvar& var, XPRBvar& var2, double val);
int setTerm(double val, XPRBvar& var, XPRBvar& var2);
int setTerm(XPRBvar& var, double val);
int setTerm(double val, XPRBvar& var);
int setTerm(double val);
    Set a term in an expression.
```

Operators

Assigning (elements to) expressions:

```
expr1 += expr2
expr1 -= expr2
expr1 = expr2
```

Composing expressions from other quadratic and linear expressions (*expr*), variables (*var*) and double values (*val*). The following operators are defined:

```
- var
- expr
expr1 + expr2
expr1 - expr2
expr * val
val * expr
var * val
val * var
var * val
var * expr
```

Throws exception 'Non-quadratic expression' if the result of the operation is not quadratic

`expr * var`
 Throws exception 'Non-quadratic expression' if the result of the operation is not quadratic

`expr1 * expr2`
 Throws exception 'Non-quadratic expression' if the result of the operation is not quadratic

Functions outside any class definition that generate quadratic expressions:

`XPRBexpr sqr(XPRBexpr& e);`
`XPRBexpr sqr(XPRBvar& var);`
 Square of an expression or variable.

Constructor detail

XPRBexpr

Synopsis

```
XPRBexpr(double d);
XPRBexpr(int i);
XPRBexpr(double d, XPRBvar& v);
XPRBexpr(double d, XPRBvar& v, XPRBvar& v2);
XPRBexpr(XPRBvar& v);
XPRBexpr(XPRBexpr& e);
```

Arguments

`d` A real value.
`i` An integer value.
`v, v2` A BCL variable.
`e` A linear or quadratic expression.

Description Create a new expression.

Method detail

add

Synopsis

```
XPRBexpr& add(XPRBexpr& e);
XPRBexpr& add(XPRBvar& v);
```

Arguments

`e` A linear or quadratic expression (may be just a constant).
`v` A BCL variable.

Return value The modified expression.

Description This method adds an expression / constant / variable to the linear or quadratic expression it is applied to.

Example See `XPRBexpr.setTerm`.

assign

Synopsis

```
XPRBexpr& assign(XPRBexpr& e);
```

Argument

`e` Expression to be copied.

Return value Copy of the expression in the argument.

Description This method copies the given expression.

delTerm

Synopsis	<pre>int delTerm(XPRBvar& var); int delTerm(XPRBvar& var, XPRBvar& var2);</pre>
Argument	var, var2 BCL decision variables.
Return value	0 if method executed successfully, 1 otherwise.
Description	This function deletes a variable term from an expression. The constant term is changed/reset with method <code>XPRBexpr.setTerm</code> .
Example	See <code>XPRBexpr.setTerm</code> .

getSol

Synopsis	<pre>double getSol();</pre>
Return value	Evaluation of the expression with the last solution.
Description	This method returns the evaluation of an expression with the solution values from the last solution found. If this method is called after completion of a global search and an integer solution has been found (that is, if method <code>XPRBprob.getMIPStat</code> returns values <code>XPRB_MIP_SOLUTION</code> or <code>XPRB_MIP_OPTIMAL</code>), it returns the value corresponding to the best integer solution. If no integer solution is available after a global search this method outputs a warning and returns 0. In all other cases it returns the evaluation corresponding to the last LP that has been solved. If this method is used <i>during</i> the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to <code>XPRBprob.sync</code> with the flag <code>XPRB_XPRS_SOL</code> .

mul

Synopsis	<pre>XPRBexpr& mul(double d); XPRBexpr& mul(XPRBexpr& e);</pre>
Arguments	d A constant. e An expression (may be just a constant or a single decision variable).
Return value	The modified expression.
Error handling	ArithmeticException 'Non-quadratic expression' if the result of the operation is not quadratic.
Description	This method multiplies an expression by a constant factor or another expression. This operation succeeds if one of the expressions is just a constant or if both expressions have only linear terms.
Example	See <code>XPRBexpr.setTerm</code> .

neg

Synopsis	<code>XPRBexpr& neg();</code>
Return value	Negation of the expression.
Description	This method multiplies an expression with -1.
Example	See <code>XPRBexpr.setTerm</code> .

setTerm

Synopsis	<pre>int setTerm(XPRBvar& var, XPRBvar& var2, double val); int setTerm(double val, XPRBvar& var, XPRBvar& var2); int setTerm(XPRBvar& var, double val); int setTerm(double val, XPRBvar& var); int setTerm(double val);</pre>
Arguments	<p><code>var, var2</code> BCL decision variables.</p> <p><code>val</code> A real value (coefficient).</p>
Return value	0 if method executed successfully, 1 otherwise.
Description	This method sets the coefficient of a variable or of the product of the two specified variables to the value <code>val</code> . If no variable is specified, the constant term of the expression is set to <code>val</code> .
Example	<p>This example shows different ways of defining and modifying a quadratic expression and finally sets the resulting expression as objective function. The comments display the definition of <code>qe</code> after each modification.</p>

```
XPRBvar x,y;
XPRBexpr qe;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);
y = prob.newVar("y", XPRB_PL, 0, 200);

qe = x;                                // x
qe.mul(3*x);                            // 3*x^2
qe += x*2*y;                            // 3*x^2 + 2*x*y
qe.add(1);                              // 1 + 3*x^2 + 2*x*y
qe.setTerm(3, x);                        // 1 + 3*x + 3*x^2 + 2*x*y
qe.setTerm(0, x, y);                    // 1 + 3*x + 3*x^2
qe.delTerm(x, x);                        // 1 + 3*x
qe.setTerm(-1);                          // - 1 + 3*x
qe -= 3*sqr(3*y);                       // - 1 + 3*x - 27*y^2
qe.neg();                                // 1 - 3*x + 27*y^2

prob.setObj(qe);
```

sqr

Synopsis

```
XPRBexpr sqr(XPRBexpr& e);  
XPRBexpr sqr(XPRBvar& var);
```

Arguments

e An expression.
var A BCL decision variable.

Return value

The square of the variable or expression in the argument.

Description

This function returns the square of the variable or expression passed in the argument if the result is at most quadratic.

Example

See `XPRBexpr.setTerm`.

XPRBIndexSet

Description

Methods for accessing index sets and operators for adding and retrieving set elements.

Constructors

```
XPRBIndexSet();  
XPRBIndexSet(xbidxset *iset);
```

Methods

```
int addElement(const char *text);  
    Add an index to an index set.  
xbidxset *getCRef();  
    Get the C modeling object.  
int getIndex(const char *text);  
    Get the index number of an index.  
const char *getIndexName(int i);  
    Get the name of an index.  
const char *getName();  
    Get the name of an index set.  
int getSize();  
    Get the size of an index set.  
bool isValid();  
    Test the validity of the index set object.  
int print();  
    Print out an index set  
void reset();  
    Reset the index set object.
```

Operators

Adding an element to an index set:

```
iset += text
```

Accessing index set elements by their name or index number:

```
int iset[text]  
const char *iset[val]
```

Constructor detail

XPRBIndexSet

Synopsis	<pre>XPRBIndexSet(); XPRBIndexSet(xbidxset *iset);</pre>
Argument	<pre>iset</pre> An index set in BCL C.
Description	Create a new index set object.

Method detail

addElement

Synopsis	<pre>int addElement(const char *text);</pre>
Argument	<pre>text</pre> Name of the index to be added to the set.
Return value	Sequence number of the index within the set, -1 in case of an error.

Description This method adds an index entry to an index set. The new element is only added to the set if no identical index already exists. Both in the case of a new index entry and an existing one, the method returns the sequence number of the index in the index set. Note that the numbering of index elements starts with 0.

Example The following example shows how to add an element to an index set and then retrieve its index and its name, (a) using the corresponding functions and (b) using the overloaded operators of this class.

```
XPRBprob prob("myprob");
XPRBindexSet ISet;
int ind;

ISet = prob.newIndexSet("IS");
ind = ISet.addElement("a");
cout << "First element: " << ISet.getIndexName(ind);
cout << ", index of 'a': " << ISet.getIndex("a") << endl;

ISet += "b";           // Add a second element
cout << "Element 1: " << ISet[1];
cout << ", index of 'b': " << ISet["b"] << endl;
```

Related topics Calls [XPRBaddidxel](#)

getCRef

Synopsis `xbidxset *getCRef();`

Return value The underlying modeling object in BCL C.

Description This method returns the index set object in BCL C that belongs to the C++ index set object.

getIndex

Synopsis `int getIndex(const char *text);`

Argument `text` Name of an index in the set.

Return value Sequence number of the index in the set, or -1 if not contained.

Description An index element can be accessed either by its name or by its sequence number. This method returns the sequence number of an index given its name.

Example See [XPRBindexSet.addElement](#).

Related topics Calls [XPRBgetidxel](#)

getIndexName

Synopsis `const char *getIndexName(int i);`

Argument `i` Index number.

Return value	Name of the i^{th} element in the set if function executed successfully, <code>NULL</code> otherwise.
Description	An index element can be accessed either by its name or by its sequence number. This method returns the name of an index set element given its sequence number.
Example	See <code>XPRBindexSet.addElement</code> .
Related topics	Calls <code>XPRBgetidxelname</code>

getName

Synopsis	<code>const char *getName();</code>
Return value	Name of the index set if function executed successfully, <code>NULL</code> otherwise.
Description	This function returns the name of an index set.
Example	See <code>XPRBindexSet.getSize</code> .
Related topics	Calls <code>XPRBgetidxsetname</code>

getSize

Synopsis	<code>int getSize();</code>
Return value	Size (= number of elements) of the set, -1 in case of an error.
Description	This function returns the current number of elements in an index set. This value does not necessarily correspond to the size specified at the creation of the set. The returned value may be smaller if fewer elements than the originally reserved number have been added, or larger if more elements have been added. (In the latter case, the size of the set is automatically increased.)
Example	<p>This example displays the name, size, and complete contents of an index set.</p> <pre> XPRBprob prob("myprob"); XPRBindexSet ISet; ISet = prob.newIndexSet("IS"); cout << ISet.getName() << " size: " << ISet.getSize() << endl; ISet.print(); </pre>
Related topics	Calls <code>XPRBgetidxsetsize</code>

isValid

Synopsis	<code>bool isValid();</code>
Return value	<code>true</code> if object is valid, <code>false</code> otherwise.
Description	This method checks whether the index set object is correctly defined. It should always be used to test the result returned by <code>XPRBprob.getIndexSetByName</code> .
Example	See <code>XPRBprob.getIndexSetByName</code> .

print

Synopsis	<code>int print();</code>
Return value	0 if function executed successfully, 1 otherwise.
Description	This method prints out an index set. It is not available in the student version.
Example	See <code>XPRBIndexSet.getSize</code> .
Related topics	Calls <code>XPRBprintidxset</code>

reset

Synopsis	<code>void reset();</code>
Description	Clear the definition of the index set object.

XPRBprob

Description

Problem definition, including methods for creating and deleting the modeling objects, problem solving, changing settings, and retrieving solution information.

Constructors

```
XPRBprob();  
XPRBprob(const char *name);
```

Methods

```
int addCuts(XPRBcut *cuts, int num);  
    Add cuts to a problem.  
void clearDir();  
    Delete all directives.  
void delCtr(XPRBctr& ctr);  
    Delete a constraint.  
void delCut(XPRBcut& cut);  
    Delete a cut definition.  
void delSos(XPRBsos& sos);  
    Delete a SOS.  
int exportProb(int format, const char *filename);  
int exportProb(int format);  
    Print problem matrix to a file.  
xbprob *getCRef();  
    Get the C modeling object.  
XPRBctr getCtrByName(const char *name);  
    Retrieve a constraint by its name.  
XPRBindexSet getIndexSetByName(const char *name);  
    Retrieve an index set by its name.  
int getLPStat();  
    Get the LP status.  
int getMIPStat();  
    Get the MIP status.  
const char *getName();  
    Get the name of the problem.  
int getNumIIS();  
    Get the number of independent IIS in an infeasible LP problem.  
double getObjVal();  
    Get the objective function value.  
int getProbStat();  
    Get the problem status.  
int getSense();  
    Get the sense of the optimization.  
XPRBsos getSosByName(const char *name);  
    Retrieve a SOS by its name.  
XPRBvar getVarByName(const char *name);  
    Retrieve a variable by its name.  
XPRSprb getXPRSprb();  
    Returns an XPRSprb problem reference for a problem defined in BCL.  
int loadBasis(const XPRBbasis& bas);  
    Load a previously saved basis.  
int loadMat();
```

Load the problem into the Xpress-Optimizer.

```
int maxim(const char *alg);
```

Maximize the objective function for the given problem.

```
int minim(const char *alg);
```

Minimize the objective function for the given problem.

```
XPRBctr newCtr(const char *name, XPRBrelation& ac);
```

```
XPRBctr newCtr(const char *name);
```

```
XPRBctr newCtr(XPRBrelation& ac);
```

```
XPRBctr newCtr();
```

Create a new constraint.

```
XPRBcut newCut(int id);
```

```
XPRBcut newCut(XPRBrelation& ac);
```

```
XPRBcut newCut(XPRBrelation& ac, int id);
```

```
XPRBcut newCut();
```

Create a new cut.

```
XPRBindexSet newIndexSet();
```

```
XPRBindexSet newIndexSet(const char *name);
```

```
XPRBindexSet newIndexSet(const char *name, int maxsize);
```

Create a new index set.

```
XPRBsos newSos(int type);
```

```
XPRBsos newSos(const char *name, int type);
```

```
XPRBsos newSos(int type, XPRBexpr& le);
```

```
XPRBsos newSos(const char *name, int type, XPRBexpr& le);
```

Create a SOS.

```
XPRBvar newVar(const char *name, int type, double lob, double upb);
```

```
XPRBvar newVar(const char *name, int type);
```

```
XPRBvar newVar(const char *name);
```

```
XPRBvar newVar();
```

Create a decision variable.

```
int print();
```

Print out the problem.

```
int printObj();
```

Print out the objective function of a problem.

```
int reset();
```

Release system resources used for storing solution information.

```
XPRBbasis saveBasis();
```

Save the current basis.

```
int setColOrder(int num);
```

Set a column ordering criterion for matrix generation.

```
int setCutMode(int mode);
```

Set the cut mode.

```
int setDictionarySize(int dict, int size);
```

Set the size of a dictionary.

```
int setMsgLevel(int lev);
```

Set the message print level.

```
int setObj(XPRBctr ctr);
```

```
int setObj(XPRBexpr e);
```

```
int setObj(XPRBvar v);
```

Select the objective function.

```
int setRealFmt(const char *fmt);
```

Set the format for printing real numbers.

```
int setSense(int dir);
    Set the sense of the optimization.
int solve(const char *alg);
    Call the Xpress-Optimizer solution algorithm.
int sync(int synctype);
    Synchronize BCL with the Optimizer.
int writeDir();
int writeDir(const char *filename);
    Write directives to a file.
```

Constructor detail

XPRBprob

Synopsis	<code>XPRBprob();</code> <code>XPRBprob(const char *name);</code>
Argument	<code>name</code> The problem name. If none specified, BCL creates a unique name.
Description	<ol style="list-style-type: none"> 1. This method needs to be called to create and initialize a new problem. If BCL has not been initialized previously this method also initializes BCL and Xpress-Optimizer. The initialization / problem creation fails if no valid license is found. 2. When solving several instances of a problem simultaneously the user must make sure to assign a different name to every instance.
Related topics	Calls <code>XPRBnewprob</code>

Method detail

addCuts

Synopsis	<code>int addCuts(XPRBcut *cuts, int num);</code>
Arguments	<code>cuts</code> Array of previously defined cuts. <code>num</code> Number of cuts in <code>cuts</code> .
Return value	0 if method executed successfully, 1 otherwise.
Description	This function adds previously defined cuts to the problem in Xpress-Optimizer. It may only be called from within the Xpress-Optimizer cut manager callback functions. BCL does not check for doubles, that is, if the user defines the same cut twice it will be added twice to the matrix. Cuts added at a node during the branch and bound search remain valid for all child nodes but are removed at all other nodes.
Example	This example show how to define the cut manager callback and add a cut to the Optimizer problem.

```
int XPRS_CC cbNode(XPRSprob oprob, void *bp)
{
    XPRBprob *bprob = (XPRBprob*)bp;
    XPRBcut aCut;
    ... // Define the cut 'aCut'
    bprob->addCuts(&aCut, 1);
    return 0; // Call this function once per node
}

int main(int argc, char **argv)
```

```

{
    XPRBprob prob("myprob");
    ...                // Define the problem
    prob.setCutMode(1);
    XPRSsetcbcutmgr(prob.getXPRSProb(), cbNode, &prob);

    prob.maxim("g");
    ...                // Solution output
}

```

Related topics Calls [XPRBaddcuts](#)

clearDir

Synopsis `void clearDir();`

Description This method deletes all directives on decision variables and SOS defined for a problem.

Example This example defines directives for a binary variable and a SOS, writes out the directives to the file `directout.dir` and then deletes all directives.

```

XPRBvar b;
XPRBsos SO2;
XPRBprob prob("myprob");
b = prob.newVar("b", XPRB_BV);

b.setDir(XPRB_UP);           // Branch upwards first
SO2.setDir(XPRB_PR, 1);      // Highest branching priority
prob.writeDir("directout");
prob.clearDir();

```

Related topics Calls [XPRBcleardir](#)

delCtr

Synopsis `void delCtr(XPRBctr& ctr);`

Argument `ctr` A BCL constraint.

Description Delete a constraint from the given problem. If this constraint has previously been selected as the objective function (using function [XPRBprob.setObj](#)), the objective will be set to `NULL`.

Related topics Calls [XPRBdelctr](#)

delCut

Synopsis `void delCut(XPRBcut& cut);`

Argument `cut` A BCL cut.

Description This method deletes the definition of a cut in BCL, but *not* the cut itself if it has already been added to the problem held in Xpress-Optimizer (using [XPRBprob.addCuts](#)).

Example See [XPRBprob.newCut](#).

Related topics Calls [XPRBdelcut](#)

delSos

Synopsis `void delSos(XPRBsos& sos);`

Argument `sos` A previously defined SOS of type 1 or 2.

Description This method deletes a SOS without deleting the variables it consists of.

Example See [XPRBprob.newSos](#).

Related topics Calls [XPRBdelsos](#)

exportProb

Synopsis `int exportProb(int format, const char *filename);`
`int exportProb(int format);`

Arguments `format` The matrix output file format, which must be one of:
 `XPRB_LP` LP file format (default);
 `XPRB_MPS` MPS file format.
`filename` Name of the output file, without extension.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This method prints the matrix to a file with an extended LP or extended MPS format. LP files receive the extension `.lp` and MPS files receive the extension `.mat`. This function is not available in the student version.
2. When exporting matrices semi-continuous and semi-continuous integer variables are preprocessed: if a lower bound value greater than 0 is given, then the variable is treated like a continuous (resp. integer) variable.

Example The following sets the sense of the optimization to maximization before exporting the problem matrix in LP format.

```
XPRBprob prob("myprob");  
prob.setSense(XPRB_MAXIM);  
prob.exportProb(XPRB_LP);
```

Related topics Calls [XPRBexportprob](#)

getCRef

Synopsis `xbprob *getCRef();`

Return value The underlying modeling object in BCL C.

Description This method returns the problem object in BCL C that belongs to the C++ problem object.

getCtrByName

Synopsis	<code>XPRBctr getCtrByName(const char *name);</code>
Argument	<code>name</code> The name of the constraint to find.
Return value	A BCL constraint.
Description	This method always returns a BCL constraint the validity of which needs to be check with <code>XPRBctr.isValid</code> . This method cannot be used if the names dictionary has been disabled (see <code>XPRBprob.setDictionarySize</code>).
Example	<p>The following retrieves a constraint by its name and if it has been found prints it out.</p> <pre>XPRBprob prob("myprob"); XPRBctr C2; C2 = prob.getCtrByName("C2"); if (C2.isValid()) C2.print();</pre>
Related topics	Calls <code>XPRBgetbyname</code>

getIndexSetByName

Synopsis	<code>XPRBIndexSet getIndexSetByName(const char *name);</code>
Argument	<code>name</code> The name of the index set to find.
Return value	A BCL index set.
Description	This method always returns a BCL index set the validity of which needs to be check with <code>XPRBIndexSet.isValid</code> . This method cannot be used if the names dictionary has been disabled (see <code>XPRBprob.setDictionarySize</code>).
Example	<p>The following retrieves an index by its name and if a set has been found prints it out.</p> <pre>XPRBprob prob("myprob"); XPRBIndexSet I2; I2 = prob.getIndexSetByName("IS"); if (I2.isValid()) I2.print();</pre>
Related topics	Calls <code>XPRBgetbyname</code>

getLPStat

Synopsis	<code>int getLPStat();</code>	
Return value	<code>0</code> <code>XPRB_LP_OPTIMAL</code> <code>XPRB_LP_INFEAS</code> <code>XPRB_LP_CUTOFF</code>	the problem has not been loaded, or error; LP optimal; LP infeasible; the objective value is worse than the cutoff;

XPRB_LP_UNFINISHED	LP unfinished;
XPRB_LP_UNBOUNDED	LP unbounded;
XPRB_LP_CUTOFF_IN_DUAL	LP cutoff in dual.
XPRB_LP_UNSOLVED	QP problem matrix is not semi-definite.

Description	The return value of this method provides LP status information from the Xpress-Optimizer.
Example	See <code>XPRBprob.solve</code> , <code>XPRBctr.getAct</code> .
Related topics	Calls <code>XPRBgetlpstat</code>

getMIPStat

Synopsis	<code>int getMIPStat();</code>	
Return value	XPRB_MIP_NOT_LOADED problem has not been loaded, or error; XPRB_MIP_LP_NOT_OPTIMAL LP has not been optimized; XPRB_MIP_LP_OPTIMAL LP has been optimized; XPRB_MIP_NO_SOL_FOUND global search incomplete — no integer solution found; XPRB_MIP_SOLUTION global search incomplete, although an integer solution has been found; XPRB_MIP_INFEAS global search complete, but no integer solution found; XPRB_MIP_OPTIMAL global search complete and an integer solution has been found.	
Description	This methods returns the global (MIP) status information from the Xpress-Optimizer.	
Example	See <code>XPRBprob.solve</code> .	
Related topics	Calls <code>XPRBgetmipstat</code>	

getName

Synopsis	<code>const char *getName();</code>	
Return value	Name of the problem if function executed successfully, <code>NULL</code> otherwise.	
Description	This method returns the problem name. If none was specified at the creation of the problem, this is a unique name created by BCL.	
Related topics	Calls <code>XPRBgetprobname</code>	

getNumIIS

Synopsis	<code>int getNumIIS();</code>	
Return value	Number of independent IIS found by Xpress-Optimizer, or a negative value in case of error.	
Description	This function returns the number of independent IIS (irreducible infeasible sets) of an infeasible LP problem.	

Related topics Calls [XPRBgetnumiis](#)

getObjVal

Synopsis `double getObjVal();`

Return value The current objective function value; default and error return value: 0.

Description This method returns the current objective function value from the Xpress-Optimizer. If it is called after completion of a global search and an integer solution has been found (that is, if [XPRBprob.getMIPStat](#) returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value of the best integer solution. In all other cases, including during a global search, it returns the solution value of the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to [XPRBprob.sync](#) with the flag `XPRB_XPRS_SOL`.

Example See [XPRBprob.solve](#).

Related topics Calls [XPRBgetobjval](#)

getProbStat

Synopsis `int getProbStat();`

Return value Bit-encoded BCL status information:

`XPRB_GEN` the matrix has been generated;
`XPRB_DIR` directives have been added;
`XPRB_MOD` the problem has been modified;
`XPRB_SOL` the problem has been solved.

Description This method returns the current BCL problem status. Note that the problem status uses bit-encoding contrary to the LP and MIP status information, because several states may apply at the same time.

Example See [XPRBprob.getXPRSprob](#).

Related topics Calls [XPRBgetprobstat](#)

getSense

Synopsis `int getSense();`

Return value `XPRB_MAXIM` the objective function is to be maximized;
`XPRB_MINIM` the objective function is to be minimized;
`-1` an error has occurred.

Description This method returns the objective sense (maximization or minimization). The sense is set to minimization by default and may be changed with [XPRBprob.setSense](#), [XPRBprob.minim](#), and [XPRBprob.maxim](#).

Related topics Calls [XPRBgetsense](#)

getSosByName

Synopsis	<code>XPRBsos getSosByName(const char *name);</code>
Argument	name The name of the SOS to find.
Return value	A BCL SOS.
Description	This method always returns a BCL SOS the validity of which needs to be check with <code>XPRBsos.isValid</code> . This method cannot be used if the names dictionary has been disabled (see <code>XPRBprob.setDictionarySize</code>).
Example	<p>The following retrieves a SOS by its name and if it has been found prints it out.</p> <pre>XPRBprob prob("myprob"); XPRBsos S2; S2 = prob.getSosByName("SO2"); if (S2.isValid()) S2.print();</pre>
Related topics	Calls <code>XPRBgetbyname</code>

getVarByName

Synopsis	<code>XPRBvar getVarByName(const char *name);</code>
Argument	name The name of the variable to find.
Return value	A BCL variable.
Description	This method always returns a BCL variable the validity of which needs to be check with <code>XPRBvar.isValid</code> . This method cannot be used if the names dictionary has been disabled (see <code>XPRBprob.setDictionarySize</code>).
Example	<p>The following retrieves a variable by its name and if it has been found prints it out.</p> <pre>XPRBprob prob("myprob"); XPRBvar b2; b2 = prob.getVarByName("b"); if (b2.isValid()) { b2.print(); cout << endl; }</pre>
Related topics	Calls <code>XPRBgetbyname</code>

getXPRSProb

Synopsis	<code>XPRSProb getXPRSProb();</code>
Return value	Reference to a problem in Xpress-Optimizer if executed successfully, <code>NULL</code> otherwise
Description	This method returns an <code>XPRSProb</code> problem reference for a problem defined in BCL and subsequently loaded into the Xpress-Optimizer. The optimizer problem may be different from the problem loaded in BCL if the solution algorithms have not been called (and the problem has not been loaded explicitly) after the last modifications

to the problem in BCL, or if any modifications have been carried out directly on the problem in the optimizer. See Section B.6 for further detail.

Example

The following example shows how to change the setting of a control parameter of Xpress-Optimizer.

```
XPRBprob bclProb("myprob");
XPRSprob optProb;
... // Define the BCL problem
if ((prob.getProbStat() & XPRB_MOD) == XPRB_MOD) prob.loadMat();
optProb = bclProb.getXPRSprob();
XPRSsetintcontrol(optProb, XPRS_PRESOLVE, 0);
```

Related topics

Calls [XPRBgetXPRSprob](#)

loadBasis

Synopsis

```
int loadBasis(const XPRBbasis& bas);
```

Argument

bas A previously saved basis.

Return value

0 if method executed successfully, 1 otherwise.

Description

This method loads a basis for the current problem. The basis must have been saved using [XPRBprob.saveBasis](#). It is *not* possible to load a basis saved for any other problem than the current one, even if the problems are similar. This function takes into account that the problem may have been modified (addition/deletion of variables and constraints) since the basis has been stored. For reading a basis from a file, the Optimizer library function [XPRSreadbasis](#) may be used. Note that the problem has to be loaded explicitly (method [XPRBprob.loadMat](#)) before the basis is re-input with [XPRBprob.loadBasis](#). Furthermore, if the reference to a basis is not used any more it should be deleted using [XPRBbasis.reset](#).

Example

See [XPRBprob.saveBasis](#).

Related topics

Calls [XPRBloadbasis](#)

loadMat

Synopsis

```
int loadMat();
```

Return value

0 if method executed successfully, 1 otherwise.

Description

This method calls the Optimizer library functions [XPRSloadlp](#), [XPRSloadqp](#), [XPRSloadglobal](#), or [XPRSloadqglobal](#) to transform the current BCL problem definition into a matrix in the Xpress-Optimizer. Empty rows and columns are deleted before generating the matrix. Semi-continuous (integer) variables are preprocessed: if a lower bound value greater than 0 is given, then the variable is treated like a continuous (resp. integer) variable. Variables that belong to the problem but do not appear in the matrix receive negative column numbers. Usually, it is *not* necessary to call this function explicitly because BCL automatically does this conversion whenever it is required. To force matrix reloading, a call to this function needs to be preceded by a call to [XPRBprob.sync](#) with the flag [XPRB_XPRS_PROB](#).

Example

See [XPRBprob.getXPRSprob](#).

Related topics

Calls [XPRBloadmat](#)

maxim

Synopsis `int maxim(const char *alg);`

Argument `alg` Choice of the solution algorithm, which should be one of:

- " " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);
- "d" solve the problem using the dual simplex algorithm;
- "p" solve the problem using the primal simplex algorithm;
- "b" solve the problem using the Newton barrier algorithm;
- "n" use the network solver (LP only);
- "l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);
- "g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value 0 if method executed successfully, 1 otherwise.

Description This method selects and starts the Xpress-Optimizer solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, e.g. "dg. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition it is regenerated automatically prior to the start of the algorithm. Matrix reloading can also be forced by calling `XPRBprob.sync` before the optimization. Before solving a problem, the objective function must be selected with `XPRBprob.setObj`. Note that if you use an incomplete global search you should finish your program with a call to the Optimizer library function `XPRSinitglobal` in order to remove all search tree information that has been stored. Otherwise you may not be able to re-run your program.

Example See `XPRBprob.solve`.

Related topics Calls `XPRBmaxim`

minim

Synopsis `int minim(const char *alg);`

Argument `alg` Choice of the solution algorithm, which should be one of:

- " " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);
- "d" solve the problem using the dual simplex algorithm;
- "p" solve the problem using the primal simplex algorithm;
- "b" solve the problem using the Newton barrier algorithm;
- "n" use the network solver (LP only);
- "l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);
- "g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value 0 if method executed successfully, 1 otherwise.

Description This method selects and starts the Xpress-Optimizer solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, e.g. "dg. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition it is regenerated automatically prior to the start of the algorithm. Matrix reloading can also be forced by calling `XPRBprob.sync` before the

optimization. Before solving a problem, the objective function must be selected with `XPRBprob.setObj`. Note that if you use an incomplete global search you should finish your program with a call to the Optimizer library function `XPRSinitglobal` in order to remove all search tree information that has been stored. Otherwise you may not be able to re-run your program.

Example See `XPRBprob.saveBasis`, `XPRBprob.solve`.

Related topics Calls `XPRBminim`

newCtr

Synopsis

```
XPRBctr newCtr(const char *name, XPRBrelation& ac);
XPRBctr newCtr(const char *name);
XPRBctr newCtr(XPRBrelation& ac);
XPRBctr newCtr();
```

Arguments

name The constraint name (of unlimited length). May be `NULL` if not required.

ac A linear or quadratic relation.

Return value A new BCL constraint.

Description This method creates a new constraint and returns the reference to this constraint, *i.e.*, the constraint's model name. If the indicated name is already in use, BCL adds an index to it. If no constraint name is given, BCL generates a default name starting with `CTR`. (The generation of unique names will only take place if the names dictionary is enabled, see `XPRBprob.setDictionarySize`.)

Example These are a few examples of constraint creation.

```
XPRBvar x,y;
XPRBctr Ctr1, Ctr2, Ctr4, Profit;
XPRBexpr le;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);
y = prob.newVar("y", XPRB_PL, 0, 200);

Ctr1 = prob.newCtr("C1", 3*x + 2*y >= 40);
Ctr2 = prob.newCtr("C2", 3*x*y + sqr(y) <= 500);
Profit = prob.newCtr("Profit", x+2*y);
prob.setObj(Profit);

le = x-5*y;
Ctr4 = prob.newCtr(le == 10);
```

Related topics Calls `XPRBnewctr`

newCut

Synopsis

```
XPRBcut newCut(int id);
XPRBcut newCut(XPRBrelation& ac);
XPRBcut newCut(XPRBrelation& ac, int id);
XPRBcut newCut();
```

Arguments

ac A linear relation defining the cut (default: equality constraint).

	<code>id</code> Cut classification or identification number (default 0).
Return value	A new BCL cut.
Description	This method creates a new cut. Cuts are loaded into the Optimizer by calling <code>XPRBprob.addCuts</code> from the Optimizer cutmanager callback.
Example	<p>The following example shows different possibilities of how to define cuts.</p> <pre> XPRBprob prob("myprob"); XPRBvar y,b; XPRBcut Cut1, Cut2, Cut3; y = prob.newVar("y", XPRB_PL, 0, 200); b = prob.newVar("b", XPRB_BV); Cut1 = prob.newCut(y == 100*b); Cut1.setID(1); Cut2 = prob.newCut(y <= 100*b, 2); Cut3 = prob.newCut(3); Cut3.setType(XPRB_L); Cut3.add(y+2); prob.delCut(Cut3); </pre>
Related topics	Calls <code>XPRBnewcut</code>

newIndexSet

Synopsis	<pre> XPRBIndexSet newIndexSet(); XPRBIndexSet newIndexSet(const char *name); XPRBIndexSet newIndexSet(const char *name, int maxsize); </pre>
Arguments	<p><code>name</code> Name of the index set to be created. May be <code>NULL</code> if not required.</p> <p><code>maxsize</code> Maximum size of the index set.</p>
Return value	A new BCL index set.
Description	This method creates a new index set. Note that the indicated size <code>maxsize</code> corresponds to the space allocated initially to the set, but it is increased dynamically if need be. If the indicated set name is already in use, BCL adds an index to it. If no name is given, BCL generates a default name starting with <code>IDX</code> . (The generation of unique names will only take place if the names dictionary is enabled, see <code>XPRBprob.setDictionarySize</code> .)
Example	<p>The following example defines an index set of size 10 and then adds two elements to the set.</p> <pre> XPRBIndexSet ISet; XPRBprob prob("myprob"); int ind; ISet = prob.newIndexSet("IS", 10); ind = ISet.addElement("a"); ISet += "b"; </pre>
Related topics	Calls <code>XPRBnewidxset</code>

newSos

Synopsis

```
XPRBsos newSos(int type);
XPRBsos newSos(const char *name, int type);
XPRBsos newSos(int type, XPRBexpr& le);
XPRBsos newSos(const char *name, int type, XPRBexpr& le);
```

Arguments

name The name of the set.

type The set type, which must be one of:

- XPRB_S1 Special Ordered Set of type 1 (default);
- XPRB_S2 Special Ordered Set of type 2.

le A linear expression.

Return value

A new BCL SOS.

Description

This method creates a Special Ordered Set (SOS) of type 1 or 2 (abbreviated SOS1 and SOS2). If the indicated name is already in use, BCL adds an index to it. If no name is given for the set, BCL generates a default name starting with `SOS`. (The generation of unique names will only take place if the names dictionary is enabled, see [XPRBprob.setDictionarySize\(\)](#).)

Example

The following example defines the SOS-1 `S01`, prints it out (output displayed as comment) and then deletes it. After this it defines an SOS-2 named `S02`.

```
XPRBvar x,y,z;
XPRBsos S01, S02;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);
y = prob.newVar("y", XPRB_PL, 0, 200);
z = prob.newVar("z", XPRB_PL, 0, 200);

S01 = prob.newSos("S01");
S01.add(x+2*y+3*z);
S01.print();           // S01(1): x(+1) y(+2) z(+3)
prob.delSos(S01);

S02 = prob.newSos("S02", XPRB_S2, 10*x+20*y);
```

Related topics

Calls [XPRBnewsos](#)

newVar

Synopsis

```
XPRBvar newVar(const char *name, int type, double lob, double upb);
XPRBvar newVar(const char *name, int type);
XPRBvar newVar(const char *name);
XPRBvar newVar();
```

Arguments

name The variable name (of unlimited length). May be `NULL` if not required.

type The variable type, which may be one of:

- XPRB_PL continuous (default);
- XPRB_BV binary;
- XPRB_UI general integer;
- XPRB_PI partial integer;
- XPRB_SC semi-continuous;
- XPRB_SI semi-continuous integer.

lob The variable's lower bound (default value: 0)
 upb The variable's upper bound (default value: XPRB_INFINITY)

Return value A new BCL decision variable.

Description

1. The creation of a variable in BCL involves not only its name but also its type and bounds. The method returns the BCL reference to the variable (*i.e.*, a model variable). If the indicated name is already in use, BCL adds an index to it. If no variable name is given, BCL generates a default name starting with `VAR`. (The generation of unique names will only take place if the names dictionary is enabled, see `XPRBprob.setDictionarySize`.) If a partial integer, semi-continuous, or semi-continuous integer variable is being created, the integer or semi-continuous limit (*i.e.* the lower bound of the continuous part for partial integer and semi-continuous, and of the semi-continuous integer part for semi-continuous integer) is set to the maximum of 1 and `bdL`. This value can be subsequently modified with the method `XPRBvar.setLim`.
2. The lower and upper bounds may take values of `-XPRB_INFINITY` and `XPRB_INFINITY` for minus and plus infinity respectively.

Example This example shows how to define different types of variables.

```
XPRBvar x,b,s;
XPRBprob prob("myprob");

x = prob.newVar("x");           // Continuous with default bounds
b = prob.newVar("b", XPRB_BV);  // Binary variable
s = prob.newVar("s", XPRB_SC, 0, 50); // Semi-cont. in [0,50]
s.setLim(10);                  // with limit value 10
```

Related topics Calls `XPRBnewvar`

print

Synopsis `int print();`

Return value 0 if function executed successfully, 1 otherwise.

Description This method prints out the complete problem definition currently held in BCL, that means, the list of constraints, any Special Ordered Sets that have been defined, and the objective function. This method is not available in the student version.

Related topics Calls `XPRBprintprob`

printObj

Synopsis `int printObj();`

Return value 0 if function executed successfully, 1 otherwise.

Description This method prints out the objective function currently defined for a problem. This method is not available in the student version.

Related topics Calls `XPRBprintobj`

reset

Synopsis `int reset();`

Return value 0 if method executed successfully, 1 otherwise.

Description This method deletes any solution information stored in BCL; it also deletes the corresponding Xpress-Optimizer problem and removes any auxiliary files that may have been created by optimization runs. It also resets the Optimizer control parameters for sparse matrix elements (`EXTRACOLS`, `EXTRAROWS`, and `EXTRAELEMENTS`) to their default values. The BCL problem definition itself remains. This method may be used to free up memory if the solution information is not required any longer but the problem definition is to be kept for later (re)use.

Related topics Calls `XPRBresetprob`

saveBasis

Synopsis `XPRBbasis saveBasis();`

Return value A BCL basis.

Description This method saves the current basis of a problem. The basis may be reinput using `XPRBprob.loadBasis`. These two methods serve for storing bases in memory; for writing a basis to a file, the Optimizer library function `XPRswritebasis` may be used. Note that there is no need to allocate space for the basis, but after its use, the basis should be deleted using `XPRBbasis.reset`. You may have to switch linear presolve and integer preprocessing off (Optimizer library controls `PRESOLVE` and `MIPPRESOLVE`) in order for the saving and reloading of bases to work correctly.

Example The following saves a basis and after some modifications to the problem reloads the problem and the saved basis into the Optimizer before re-solving the problem.

```
XPRBbasis bas;
XPRBprob prob("myprob");

bas = prob.saveBasis();
...           // Modify the problem
prob.loadMat();
prob.loadBasis(bas);
bas.reset();
prob.minim();
```

Related topics Calls `XPRBsavebasis`

setColOrder

Synopsis `int setColOrder(int num);`

Argument `num` The ordering flag, which must be one of:
0 default ordering;
1 alphabetical order.

Return value 0 if method executed successfully, 1 otherwise.

- Description**
1. BCL runs reproduce always the same matrix for a problem. This method allows the user to choose a different ordering criterion than the default one. Note that this method only changes the order of columns in what is sent to Xpress-Optimizer, you do not see any effect when exporting the matrix with BCL. However you can control the effect by exporting the matrix from the Optimizer.
 2. To change this setting for all problems that are created subsequently use the corresponding method of class `XPRB`.

Related topics Calls `XPRBsetcolorder`

setCutMode

Synopsis `int setCutMode(int mode);`

Argument `mode` Cut mode indicator:
 0 switch cut mode off
 1 switch cut mode on

Return value 0 if method executed successfully, 1 otherwise.

Description This function switches the cut mode on or off. It changes the settings of certain Optimizer controls. Switching the cut mode off resets these controls to their default values.

Example See `XPRBprob.addCuts`.

Related topics Calls `XPRBsetcutmode`

setDictionarySize

Synopsis `int setDictionarySize(int dict, int size);`

Arguments `dict` Choice of the dictionary. Possible values:
 `XPRB_DICT_NAMES` names dictionary
 `XPRB_DICT_IDX` indices dictionary

 `size` Non-negative value, preferably a prime number; 0 disables the dictionary (for names dictionary only).

Return value 0 if method executed successfully, 1 otherwise.

Description

This function sets the size of the hash table of the names or indices dictionaries (defaults: names 2999, indices 1009) of the given problem. It can only be called immediately after the creation of the corresponding problem.

The names dictionary serves for storing and accessing the names of all modeling objects (variables, arrays of variables, constraints, SOS, index sets). Once it has been disabled it cannot be enabled any more. All methods relative to the names cannot be used if this dictionary has been disabled and BCL will not generate any unique names at the creation of model objects. If you wish to use the names dictionary we recommend to choose a size close to the number of variables+constraints in your problem, preferably a prime number.

The indices dictionary serves for storing all index set elements. The indices dictionary cannot be disabled, it is created automatically once an index set element is defined.

Related topics Calls `XPRBsetdictionarysize`

setMsgLevel

Synopsis	<code>int setMsgLevel(int lev);</code>												
Argument	<table><tr><td>level</td><td>The message level, <i>i.e.</i> the type of messages printed by BCL. This may be one of:</td></tr><tr><td>0</td><td>no messages printed;</td></tr><tr><td>1</td><td>error messages only printed;</td></tr><tr><td>2</td><td>warnings and errors printed;</td></tr><tr><td>3</td><td>warnings, errors, and Optimizer log printed (default);</td></tr><tr><td>4</td><td>all messages printed.</td></tr></table>	level	The message level, <i>i.e.</i> the type of messages printed by BCL. This may be one of:	0	no messages printed;	1	error messages only printed;	2	warnings and errors printed;	3	warnings, errors, and Optimizer log printed (default);	4	all messages printed.
level	The message level, <i>i.e.</i> the type of messages printed by BCL. This may be one of:												
0	no messages printed;												
1	error messages only printed;												
2	warnings and errors printed;												
3	warnings, errors, and Optimizer log printed (default);												
4	all messages printed.												
Return value	0 if method executed successfully, 1 otherwise.												
Description	<ol style="list-style-type: none">1. BCL can produce different types of messages; status information, warnings and errors. This function controls which of these are output. For settings 1 or higher, the corresponding Optimizer output is also displayed. In addition to this setting, the amount of Optimizer output can be modified through several Optimizer printing control parameters (see the 'Xpress-Optimizer Reference Manual').2. To change this setting for all problems that are created subsequently use the corresponding method of class <code>XPRB</code>.												
Example	<p>The following example changes the global BCL message printing level to 'errors' only and sets the printing level for problem <code>prob</code> back to the default. It also modifies the values of the Optimizer printing controls for simplex and MIP logging.</p> <pre>XPRBprob prob("myprob"); XPRB::setMsgLevel(1); prob.setMsgLevel(3); XPRSsetintcontrol(prob.getXPRSprob(), XPRS_LPLOG, 0); XPRSsetintcontrol(prob.getXPRSprob(), XPRS_MIPLLOG, -500);</pre>												
Related topics	Calls <code>XPRBsetmsglevel</code>												

setObj

Synopsis	<pre>int setObj(XPRBctr ctr); int setObj(XPRBexpr e); int setObj(XPRBvar v);</pre>						
Arguments	<table><tr><td>ctr</td><td>A BCL constraint.</td></tr><tr><td>e</td><td>A linear or quadratic expression.</td></tr><tr><td>v</td><td>A BCL decision variable.</td></tr></table>	ctr	A BCL constraint.	e	A linear or quadratic expression.	v	A BCL decision variable.
ctr	A BCL constraint.						
e	A linear or quadratic expression.						
v	A BCL decision variable.						
Return value	0 if method executed successfully, 1 otherwise.						
Description	This functions sets the objective function by selecting a constraint the variable terms of which become the objective function. This must be done before any optimization task is carried out. Typically, the objective constraint will have the type <code>XPRB_N</code> (non-binding), but any other type of constraint may be chosen too. In the latter case, the equation or inequality expressed by the constraint also remains part of the problem.						
Example	See <code>XPRBprob.newCtr</code> .						
Related topics	Calls <code>XPRBsetobj</code>						

setRealFmt

Synopsis	<code>int setRealFmt(const char *fmt);</code>						
Argument	<table><tr><td><code>fmt</code></td><td>Format string (as used by the C function <code>printf</code>). Simple format strings are of the form <code>%n</code> where <code>n</code> may be, for instance, one of</td></tr><tr><td><code>g</code></td><td>default printing format (precision: 6 digits; exponential notation if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision)</td></tr><tr><td><code>.numf</code></td><td>print real numbers in the style <code>[-]d.d</code> where the number of digits after the decimal point is equal to the given precision <code>num</code>.</td></tr></table>	<code>fmt</code>	Format string (as used by the C function <code>printf</code>). Simple format strings are of the form <code>%n</code> where <code>n</code> may be, for instance, one of	<code>g</code>	default printing format (precision: 6 digits; exponential notation if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision)	<code>.numf</code>	print real numbers in the style <code>[-]d.d</code> where the number of digits after the decimal point is equal to the given precision <code>num</code> .
<code>fmt</code>	Format string (as used by the C function <code>printf</code>). Simple format strings are of the form <code>%n</code> where <code>n</code> may be, for instance, one of						
<code>g</code>	default printing format (precision: 6 digits; exponential notation if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision)						
<code>.numf</code>	print real numbers in the style <code>[-]d.d</code> where the number of digits after the decimal point is equal to the given precision <code>num</code> .						
Return value	0 if method executed successfully, 1 otherwise.						
Description	<ol style="list-style-type: none">1. In problems with very large or very small numbers it may become necessary to change the printing format to obtain a more exact output. In particular, by changing the precision it is possible to reduce the difference between matrices loaded in memory into Xpress-Optimizer and the output produced by exporting them to a file.2. To change this setting for all problems that are created subsequently use the corresponding method of class <code>XPRB</code>.						
Example	See <code>XPRB.setRealFmt</code> .						
Related topics	Calls <code>XPRBsetrealfmt</code>						

setSense

Synopsis	<code>int setSense(int dir);</code>						
Argument	<table><tr><td><code>dir</code></td><td>Sense of the objective function, which must be one of:</td></tr><tr><td><code>XPRB_MAXIM</code></td><td>maximize the objective;</td></tr><tr><td><code>XPRB_MINIM</code></td><td>minimize the objective.</td></tr></table>	<code>dir</code>	Sense of the objective function, which must be one of:	<code>XPRB_MAXIM</code>	maximize the objective;	<code>XPRB_MINIM</code>	minimize the objective.
<code>dir</code>	Sense of the objective function, which must be one of:						
<code>XPRB_MAXIM</code>	maximize the objective;						
<code>XPRB_MINIM</code>	minimize the objective.						
Return value	0 if method executed successfully, 1 otherwise.						
Description	This method sets the optimization sense to maximization or minimization. It is set to minimization by default.						
Example	See <code>XPRBprob.exportProb</code> .						
Related topics	Calls <code>XPRBsetsense</code>						

solve

Synopsis	<code>int solve(const char *alg);</code>
-----------------	------------------------------------------

Argument	<div>alg</div> <div>Choice of the solution algorithm, which should be one of:</div> <div> " " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state); "d" solve the problem using the dual simplex algorithm; "p" solve the problem using the primal simplex algorithm; "b" solve the problem using the Newton barrier algorithm; "n" use the network solver (LP only); "l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved); "g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved. </div>
Return value	0 if method executed successfully, 1 otherwise.
Description	<p>This method selects and starts the Xpress-Optimizer solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, e.g. "dg". If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition it is regenerated automatically prior to the start of the algorithm. Matrix reloading can also be forced by calling <code>XPRBprob.sync</code> before the optimization. The sense of the optimization (default: minimization) can be changed with function <code>XPRBprob.setSense</code>. Before solving a problem, the objective function must be selected with <code>XPRBprob.setObj</code>. Note that if you use an incomplete global search you should finish your program with a call to the Optimizer library function <code>XPRSinitglobal</code> in order to remove all search tree information that has been stored. Otherwise you may not be able to re-run your program.</p>
Example	<p>The following example first maximizes the LP relaxation of a problem and then solves the problem as a MIP. After each optimization run the objective function value is displayed.</p> <pre> XPRBprob prob("myprob"); ... // Define the problem prob.setSense(XPRB_MAXIM); prob.solve(""); if (prob.getLPStat() == XPRB_LP_OPTIMAL) cout << "LP solution: " << prob.getObjVal() << endl; prob.solve("g"); if (prob.getMIPStat() == XPRB_MIP_OPTIMAL prob.getMIPStat() == XPRB_MIP_SOLUTION) cout << "MIP solution: " << prob.getObjVal() << endl; </pre>
Related topics	Calls <code>XPRBsolve</code>

sync

Synopsis	<code>int sync(int synctype);</code>
Argument	<div>synctype</div> <div>Type of the synchronization. Possible values:</div> <div> XPRB_XPRS_SOL update the BCL solution information with the solution currently held in the Optimizer; XPRB_XPRS_PROB force problem reloading. </div>
Return value	0 if method executed successfully, 1 otherwise.

- Description**
1. This method resets the BCL problem status.
 2. `XPRB_XPRS_SOL`: at the next solution access the solution information in BCL is updated with the solution held in the Optimizer (after MIP search: best integer solution, otherwise solution of the last LP solved).
 3. `XPRB_XPRS_PROB`: at the next call to optimization or `XPRBloadmat` the problem is completely reloaded into the Optimizer; bound changes are not passed on to the problem loaded in the Optimizer any longer.

Example The following forces BCL to reload the matrix into the Optimizer even if there has been no change other than bound changes to the problem definition in BCL since the preceding optimization / matrix loading.

```
XPRBprob prob("myprob");
...           // Define + load the problem
prob.sync(XPRB_XPRS_PROB);
prob.solve("g");
```

Related topics Calls `XPRBsync`

writeDir

- Synopsis** `int writeDir();`
`int writeDir(const char *filename);`
- Argument** `filename` Name of the directives files.
- Return value** 0 if method executed successfully, 1 otherwise.
- Description** This method writes out to a file the directives defined for a problem. If the given file name does not include an extension the extension `.dir` is appended to it. When no file name is given, the name of the problem is used. If a file of the given name exists already it is replaced.
- Example** See `XPRBprob.clearDir`.
- Related topics** Calls `XPRBwritedir`

XPRBrelation (extends XPRBexpr)

Description

Methods and operators for constructing linear or quadratic relations from expressions.

Constructors

```
XPRBrelation(const XPRBexpr& e, int type);  
XPRBrelation(const XPRBexpr& e);  
XPRBrelation(const XPRBvar& v);
```

Methods

```
int getType();  
    Get the relation type.
```

Operators

Creating relations by establishing relations between linear or quadratic expressions. The following operators are defined outside any class definition:

```
expr1 <= expr2  
expr1 >= expr2  
expr1 == expr2
```

Constructor detail

XPRBrelation

Synopsis

```
XPRBrelation(const XPRBexpr& e, int type);  
XPRBrelation(const XPRBexpr& e);  
XPRBrelation(const XPRBvar& v);
```

Arguments

e	A linear or quadratic expression.
type	The relation type, which must be one of: XPRB_L 'less than or equal to' constraint; XPRB_G 'greater than or equal to' constraint; XPRB_E an equality; XPRB_N a non-binding row (default).
v	A BCL variable.

Description

Create a new linear or quadratic relation.

Method detail

getType

Synopsis

```
int getType();
```

Return value

XPRB_L	'less than or equal to' inequality;
XPRB_G	'greater than or equal to' inequality;
XPRB_E	equality;
XPRB_N	a non-binding row (objective function);
-1	an error has occurred.

Description

This method returns the relation type if successful, and -1 in case of an error.

XPRBsos

Description

Methods for modifying and accessing Special Ordered Sets and operators for constructing them.

Constructors

```
XPRBsos();  
XPRBsos(xbsos *s);  
XPRBsos(xbsos *s, XPRBexpr& l);
```

Methods

```
void add(const XPRBexpr& le);  
    Add a linear expression to a SOS.  
int addElement(XPRBvar& var, double val);  
int addElement(double val, XPRBvar& var);  
    Add an element to a SOS.  
int delElement(XPRBvar& var);  
    Delete an element from a SOS.  
xbsos *getCRef();  
    Get the C modeling object.  
const char *getName();  
    Get the name of a SOS.  
int getType();  
    Get the type of a SOS.  
bool isValid();  
    Test the validity of the SOS object.  
int print();  
    Print out a SOS  
int setDir(int type, double val);  
int setDir(int type);  
    Set a branching directive for a SOS.
```

Operators

Assigning and adding linear expressions to Special Ordered Sets:

```
set = linexp  
set += linexp
```

Constructor detail

XPRBsos

Synopsis

```
XPRBsos();  
XPRBsos(xbsos *s);  
XPRBsos(xbsos *s, XPRBexpr& l);
```

Arguments

s A SOS in BCL C.
l Linear expression defining the SOS.

Description

Create a new SOS object.

Method detail

add

Synopsis	<code>void add(const XPRBexpr& le);</code>
Argument	<code>le</code> A linear expression.
Description	This method adds the variables of a linear expression to a SOS, using their coefficients in the linear expression as weights.
Example	This example shows different ways of defining SOS and modifying their contents. The resulting SOS definitions (as obtained with <code>XPRBsos.print</code>) and the output printed by the program are displayed as comments.

```
XPRBvar x,y,z;
XPRBsos S01, S02;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);
y = prob.newVar("y", XPRB_PL, 0, 200);
z = prob.newVar("z", XPRB_PL, 0, 200);

S01 = prob.newSos("S01", XPRB_S1);
S01.add(x+2*y+3*z);           // S01(1): x(+1) y(+2) z(+3)
S01 += 2*z-x;                 // S01(1): y(+2) z(+5)
cout << S01.getName() << " type: ";
cout << (S01.getType()==XPRB_S1?1:2) << endl;
                               // S01 type: 1
S02 = prob.newSos("S02", XPRB_S2, 10*x+20*y);
S02.addElement(z, 5);         // S02(2): x(+10) y(+20) z(+5)
S02.delElement(x);            // S02(2): y(+20) z(+5)
```

addElement

Synopsis	<code>int addElement(XPRBvar& var, double val);</code> <code>int addElement(double val, XPRBvar& var);</code>
Arguments	<code>var</code> Reference to a variable. <code>val</code> The corresponding weight or reference value.
Return value	0 if function executed successfully, 1 otherwise
Description	This method adds a single variable and its weight coefficient to a Special Ordered Set. If the variable is already contained in the set, the indicated value is added to its weight. Note that weight coefficients must be different from 0.
Example	See <code>XPRBsos.add</code> .
Related topics	Calls <code>XPRBaddsosel</code>

delElement

Synopsis	<code>int delElement(XPRBvar& var);</code>
-----------------	------------------------------------------------

Argument	<code>var</code> A BCL variable.
Return value	0 if method executed successfully, 1 otherwise.
Description	This function removes a variable from a Special Ordered Set.
Example	See <code>XPRBsos.add</code> .
Related topics	Calls <code>XPRBdelssel</code>

getCRef

Synopsis	<code>xbsos *getCRef();</code>
Return value	The underlying modeling object in BCL C.
Description	This method returns the SOS object in BCL C that belongs to the C++ SOS object.

getName

Synopsis	<code>const char *getName();</code>
Return value	Name of the SOS if executed successfully, <code>NULL</code> otherwise.
Description	This method returns the name of a SOS. If the user has not defined a name the default name generated by BCL is returned.
Example	See <code>XPRBsos.add</code> .
Related topics	Calls <code>XPRBgetsosname</code>

getType

Synopsis	<code>int getType();</code>
Return value	<code>XPRB_S1</code> a Special Ordered Set of type 1; <code>XPRB_S2</code> a Special Ordered Set of type 2; <code>-1</code> an error has occurred.
Description	This method returns the type of a SOS.
Example	See <code>XPRBsos.add</code> .
Related topics	Calls <code>XPRBgetsostype</code>

isValid

Synopsis	<code>bool isValid();</code>
Return value	<code>true</code> if object is valid, <code>false</code> otherwise.

Description	This method checks whether the SOS object is correctly defined. It should always be used to test the result returned by <code>XPRBprob.getSosByName</code> .
Example	See <code>XPRBprob.getSosByName</code> .

print

Synopsis	<code>int print();</code>
Return value	0 if function executed successfully, 1 otherwise.
Description	This method prints out a SOS. It is not available in the student version.
Example	See <code>XPRBprob.getSosByName</code> .
Related topics	Calls <code>XPRBprintsos</code>

setDir

Synopsis	<code>int setDir(int type, double val);</code> <code>int setDir(int type);</code>
Arguments	<div> <div> type <p>The directive type, which must be one of:</p> <ul style="list-style-type: none"> <code>XPRB_PR</code> priority; <code>XPRB_UP</code> first branch upwards; <code>XPRB_DN</code> first branch downwards; <code>XPRB_PU</code> pseudo cost on branching upwards; <code>XPRB_PD</code> pseudo cost on branching downwards. </div> <div> val <p>An argument dependent on the type of the directive being defined. If <code>type</code> is:</p> <ul style="list-style-type: none"> <code>XPRB_PR</code> <code>val</code> will be the priority value, an integer between 1 (highest) and 1000 (lowest), the default; <code>XPRB_UP</code> no input is required; <code>XPRB_DN</code> no input is required; <code>XPRB_PU</code> <code>val</code> will be the value of the pseudo cost for the upward branch; <code>XPRB_PD</code> <code>val</code> will be the value of the pseudo cost for the downward branch. </div> </div>
Return value	0 if method executed successfully, 1 otherwise.
Description	This method sets any type of branching directive available in Xpress-MP. This may be a priority for branching on a SOS (type <code>XPRB_PR</code>), the preferred branching direction (types <code>XPRB_UP</code> , <code>XPRB_DN</code>) or the estimated cost incurred when branching on a SOS (types <code>XPRB_PU</code> , <code>XPRB_PD</code>). Several directives of different types may be set for a single set. Method <code>XPRBvar.setDir</code> may be used to set a directive for a variable.
Example	See <code>XPRBprob.clearDir</code> .
Related topics	Calls <code>XPRBsetsosdir</code>

XPRBvar

Description

Methods for modifying and accessing variables.

Constructors

```
XPRBvar();  
XPRBvar(xbvar *v);
```

Methods

```
int fix(double val);  
    Fix a variable.  
int getColNum();  
    Get the column number for a variable.  
xbvar *getCRef();  
    Get the C modeling object.  
double getLB();  
    Get the lower bound on a variable.  
double getLim();  
    Get the integer limit for a partial integer or the semi-continuous limit for a semi-  
    continuous or semi-continuous integer variable.  
const char *getName();  
    Get the name of a variable.  
double getRCost();  
    Get the reduced cost value.  
double getRNG(int rngtype);  
    Get ranging information.  
double getSol();  
    Get the solution value.  
int getType();  
    Get the type of a variable.  
double getUB();  
    Get the upper bound on a variable.  
bool isValid();  
    Test the validity of the variable object.  
int print();  
    Print out a variable.  
int setDir(int type, double val);  
int setDir(int type);  
    Set a branching directive for a variable.  
int setLB(double val);  
    Set a lower bound.  
int setLim(double val);  
    Set the integer limit for a partial integer, or the lower semi-continuous limit for a  
    semi-continuous or semi-continuous integer variable.  
int setType(int type);  
    Set the variable type.  
int setUB(double val);  
    Set an upper bound.
```

Constructor detail

XPRBvar

Synopsis	<code>XPRBvar();</code> <code>XPRBvar(xbvar *v);</code>
Argument	<code>v</code> A variable in BCL C.
Description	Create a new variable object.

Method detail

fix

Synopsis	<code>int fix(double val);</code>
Argument	<code>val</code> The value to which the variable is to be fixed.
Return value	0 if method executed successfully, 1 otherwise.
Description	This method fixes a variable to the given value. It replaces calls to <code>XPRBvar.setLB</code> and <code>XPRBvar.setUB</code> . The value <code>val</code> may lie outside the original bounds of the variable. If the problem is loaded in the Optimizer, the bound change is passed on immediately without any need to reload the problem.
Related topics	Calls <code>XPRBfixvar</code>

getColNum

Synopsis	<code>int getColNum();</code>
Return value	Column number (non-negative value), or a negative value.
Description	This method returns the column number of a variable in the matrix currently loaded in the Xpress-Optimizer. If the variable is not part of the matrix, or if the matrix has not yet been generated, the function returns a negative value. To check whether the matrix has been generated, use function <code>XPRBprob.getProbStat</code> . The counting of column numbers starts with 0.
Example	See <code>XPRBvar.getSol</code> .
Related topics	Calls <code>XPRBgetcolnum</code>

getCRef

Synopsis	<code>xbvar *getCRef();</code>
Return value	The underlying modeling object in BCL C.
Description	This method returns the variable object in BCL C that belongs to the C++ variable object.

getLB

Synopsis	<code>double getLB();</code>
Return value	Lower bound on the variable (default 0).
Description	This method returns the currently defined lower bound on a variable.
Example	See <code>XPRBvar.getName</code> .
Related topics	Calls <code>XPRBgetbounds</code>

getLim

Synopsis	<code>double getLim();</code>
Return value	Limit value (default 1):
Description	This method returns the currently defined integer limit for a partial integer variable or the lower semi-continuous limit for a semi-continuous or semi-continuous integer variable.
Example	See <code>XPRBvar.getName</code> .
Related topics	Calls <code>XPRBgetlim</code>

getName

Synopsis	<code>const char *getName();</code>
Return value	Name of the variable if executed successfully, <code>NULL</code> otherwise.
Description	This method returns the name of a variable. If the user has not defined a name the default name generated by BCL is returned.
Example	<p>The following example displays information about a semi-continuous variable. The output printed by this program extract is shown in the comment.</p> <pre>XPRBvar s; XPRBprob prob("myprob"); s = prob.newVar("s", XPRB_SC, 0, 200); s.setLim(10); if (s.getType()==XPRB_SC s.getType()==XPRB_SI) { cout << s.getName() << " in {" << s.getLB() << "}" +["; cout << s.getLim() << "," << s.getUB() << "]" << endl; } // s in {0}+[10,200]</pre>
Related topics	Calls <code>XPRBgetvarname</code>

getRCost

Synopsis `double getRCost();`

Return value Reduced cost value for the variable, 0 in case of an error.

Description This method returns the reduced cost value for a variable. The user may wish to test first whether this variable is part of the problem, for instance by checking that the column number is non-negative.
If this function is called after completion of a global search and an integer solution has been found (that is, if function `XPRBprob.getMIPStat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the reduced cost value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBprob.sync` with the flag `XPRB_XPRS_SOL`.

Example See `XPRBvar.getSol`.

Related topics Calls `XPRBgetrcost`

getRNG

Synopsis `double getRNG(int rngtype);`

Argument `rngtype` The type of ranging information sought. This is one of:

<code>XPRB_UPACT</code>	upper activity;
<code>XPRB_LOACT</code>	lower activity;
<code>XPRB_UUP</code>	upper unit cost;
<code>XPRB_UDN</code>	lower unit cost
<code>XPRB_UCOST</code>	upper cost;
<code>XPRB_LCOST</code>	lower cost.

Return value Ranging information of the required type.

Description This method can only be used after solving an LP problem. Ranging information for MIP problems can be obtained by fixing all discrete variables to their solution values and re-solving the resulting LP problem.

Example This example retrieves ranging information (lower and upper activity) for a variable.

```
XPRBvar x;
XPRBprob prob("myprob");

x = prob.newVar("x", XPRB_PL, 0, 200);

... // Define and solve an LP problem

cout << "x: " << x.getSol();
cout << " (act. range: " << x.getRNG(XPRB_LOACT) << ", " ;
cout << x.getRNG(XPRB_UPACT) << ")" << endl;
```

Related topics Calls `XPRBgetvarrng`

getSol

Synopsis `double getSol();`

Return value Primal solution value for the variable, 0 in case of an error.

Description This function returns the current solution value for a variable. The user may wish to test first whether this variable is part of the problem, for instance by checking that the column number is non-negative.
If this function is called after completion of a global search and an integer solution has been found (that is, if function `XPRBprob.getMIPStat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value of the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the solution value in the last LP that has been solved. If this function is used *during* the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBprob.sync` with the flag `XPRB_XPRS_SOL`.

Example This example retrieves the solution information for the variable `x` after solving an LP problem.

```
XPRBprob prob("myprob");
XPRBvar x;
...
x = prob.newVar("x", XPRB_PL, 0, 200);
prob.solve("l");
if (x.getColNum() >= 0 && prob.getLPStat() == XPRB_LP_OPTIMAL)
{
    cout << x.getName() << ": solution: " << x.getSol();
    cout << " reduced cost: " << x.getRCost() << endl;
}
else
    cout << "No solution information available." << endl;
```

Related topics Calls `XPRBgetsol`

getType

Synopsis `int getType();`

Return value

<code>XPRB_PL</code>	continuous;
<code>XPRB_BV</code>	binary;
<code>XPRB_UI</code>	general integer;
<code>XPRB_PI</code>	partial integer;
<code>XPRB_SC</code>	semi-continuous;
<code>XPRB_SI</code>	semi-continuous integer;
<code>-1</code>	an error has occurred.

Description If the function exits successfully, the variable type is returned.

Example See `XPRBvar.getName`.

Related topics Calls `XPRBgetvartype`

getUB

Synopsis	<code>double getUB();</code>
Return value	Upper bound on the variable (default <code>XPRB_INFINITY</code>).
Description	This method returns the currently defined upper bound on a variable.
Example	See <code>XPRBvar.getName</code> .
Related topics	Calls <code>XPRBgetbounds</code>

isValid

Synopsis	<code>bool isValid();</code>
Return value	<code>true</code> if object is valid, <code>false</code> otherwise.
Description	This method checks whether the variable object is correctly defined. It should always be used to test the result returned by <code>XPRBprob.getVarByName</code> .
Example	See <code>XPRBprob.getVarByName</code> .

print

Synopsis	<code>int print();</code>
Return value	The number of characters printed.
Description	This method prints out a variable. It is not available in the student version.
Example	See <code>XPRBprob.getVarByName</code> .
Related topics	Calls <code>XPRBprintvar</code>

setDir

Synopsis	<code>int setDir(int type, double val);</code> <code>int setDir(int type);</code>												
Arguments	<table><tr><td><code>type</code></td><td>Directive type, which must be one of:</td></tr><tr><td><code>XPRB_PR</code></td><td>priority;</td></tr><tr><td><code>XPRB_UP</code></td><td>first branch upwards;</td></tr><tr><td><code>XPRB_DN</code></td><td>first branch downwards;</td></tr><tr><td><code>XPRB_PU</code></td><td>pseudo cost on branching upwards;</td></tr><tr><td><code>XPRB_PD</code></td><td>pseudo cost on branching downwards.</td></tr></table>	<code>type</code>	Directive type, which must be one of:	<code>XPRB_PR</code>	priority;	<code>XPRB_UP</code>	first branch upwards;	<code>XPRB_DN</code>	first branch downwards;	<code>XPRB_PU</code>	pseudo cost on branching upwards;	<code>XPRB_PD</code>	pseudo cost on branching downwards.
<code>type</code>	Directive type, which must be one of:												
<code>XPRB_PR</code>	priority;												
<code>XPRB_UP</code>	first branch upwards;												
<code>XPRB_DN</code>	first branch downwards;												
<code>XPRB_PU</code>	pseudo cost on branching upwards;												
<code>XPRB_PD</code>	pseudo cost on branching downwards.												

<code>val</code>	An argument dependent on the type of directive to be defined. Must be one of:
<code>XPRB_PR</code>	priority value — an integer between 1 (highest) and 1000 (least priority), the default;
<code>XPRB_UP</code>	no input required;
<code>XPRB_DN</code>	no input required;
<code>XPRB_PU</code>	value of the pseudo cost on branching upwards;
<code>XPRB_PD</code>	value of the pseudo cost on branching downwards.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This method sets any type of branching directive available in Xpress-MP. This may be a priority for branching on a variable (type `XPRB_PR`), the preferred branching direction (types `XPRB_UP`, `XPRB_DN`) or the estimated cost incurred when branching on a variable (types `XPRB_PU`, `XPRB_PD`). Several directives of different types may be set for a single variable.
2. Note that it is only possible to set branching directives for discrete variables (including semi-continuous and partial integer variables). Method `XPRBsos.setDir` may be used to set a directive for a SOS.

Example See `XPRBprob.clearDir`.

Related topics Calls `XPRBsetvardir`

setLB

Synopsis `int setLB(double val);`

Argument `val` The variable's new lower bound.

Return value 0 if method executed successfully, 1 otherwise.

Description This method sets the lower bound on a variable. If the problem is loaded in the Optimizer, the bound change is passed on immediately without any need to reload the problem.

Related topics Calls `XPRBsetlb`

setLim

Synopsis `int setLim(double val);`

Argument `val` Value of the integer limit.

Return value 0 if method executed successfully, 1 otherwise.

Description This method sets the integer limit (*i.e.* the lower bound of the continuous part) of a partial integer variable or the semi-continuous limit of a semi-continuous or semi-continuous integer variable to the given value.

Example See `XPRBvar.getName`, `XPRBprob.newVar`.

Related topics Calls `XPRBsetlim`

setType

Synopsis `int setType(int type);`

Argument `type` The variable type, which is one of:

<code>XPRB_PL</code>	continuous;
<code>XPRB_BV</code>	binary;
<code>XPRB_UI</code>	general integer;
<code>XPRB_PI</code>	partial integer;
<code>XPRB_SC</code>	semi-continuous;
<code>XPRB_SI</code>	semi-continuous integer.

Return value 0 if method executed successfully, 1 otherwise.

Description This method changes the type of a variable that has been created previously.

Related topics Calls [XPRBsetvartype](#)

setUB

Synopsis `int setUB(double val);`

Argument `val` The variable's new upper bound.

Return value 0 if method executed successfully, 1 otherwise.

Description This method sets the upper bound on a variable. If the problem is loaded in the Optimizer, the bound change is passed on immediately without any need to reload the problem.

Related topics Calls [XPRBsetub](#)

Chapter 6

BCL in Java

6.1 An overview of BCL in Java

Much as for the C++ interface, the Java interface of BCL provides the full functionality of the C version except for the data input, output and error handling for which the standard Java system functions can be used. The C modeling objects, such as variables, constraints and problems, are again converted into classes, and their associated functions into methods of the corresponding class in Java.

Whereas in C++ it is possible to use C functions, such as `printf` or `XPRBprintf` for printing output, all code in Java programs must be written in Java itself. In addition, in Java it is not possible to overload the algebraic operators as has been done for the definition of constraints in C++. Instead, the Java interface provides a set of simple methods like `add` or `eq1` that have been overloaded to accept various types and numbers of parameters.

The names for classes and methods in Java have been formed in the same way as those of their counterparts in C++: All Java classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsos`, `XPRBindexSet`, `XPRBbasis`) take the same names, with the exception of `XPRBindexSet`. In the names of the methods the prefix `XPRB` has been dropped, as have references to the type of the object. For example, function `XPRBgetvarname` is turned into the method `getName` of class `XPRBvar`.

All Java BCL classes are contained in the package `com.dashoptimization`. To use the (short) class names, it is recommended to add the line

```
import com.dashoptimization.*;
```

at the beginning of every program that uses the Java classes of BCL.

The C++ classes and their methods documented in section 5.2 correspond to a large extent to the classes defined by the Java interface, with some additional classes in the Java version. A comprehensive documentation of the BCL Java interface is available as a separate '[Java on-line documentation](#)'.

6.1.1 Example

An example of use of BCL in Java is the following, which again constructs the example described in Chapter 2. Contrary to the C and C++ versions, BCL Java needs to be initialized explicitly by creating an instance of `XPRB`.

```
import com.dashoptimization.*;

public class xbexpl1
{
    static final int NJ = 4;           /* Number of jobs */
    static final int NT = 10;          /* Time limit */

    static final double[] DUR = {3,4,2,2}; /* Durations of jobs */
}
```

```

static XPRBvar[] start;          /* Start times of jobs */
static XPRBvar[][] delta;       /* Binaries for start times */
static XPRBvar z;               /* Max. completion time */

static XPRB bcl;
static XPRBprob p;

static void jobsModel()
{
    XPRBexpr le;
    int j,t;

    start = new XPRBvar[NJ];      /* Start time variables */
    for(j=0;j<NJ;j++) start[j] = p.newVar("start");
    z = p.newVar("z",XPRB.PL,0,NT); /* Makespan variable */

    delta = new XPRBvar[NJ][NT];
    for(j=0;j<NJ;j++)            /* Binaries for each job */
        for(t=0;t<(NT-DUR[j]+1);t++)
            delta[j][t] = p.newVar("delta"+(j+1)+(t+1), XPRB.BV);

    for(j=0;j<NJ;j++)            /* Calculate max. completion time */
        p.newCtr("Makespan", start[j].add(DUR[j]).lEq(z));

    p.newCtr("Prec", start[0].add(DUR[0]).lEq(start[2]));
                                /* Precedence rel. between jobs */

    for(j=0;j<NJ;j++)            /* Linking start times & binaries */
    {
        le = new XPRBexpr();
        for(t=0;t<(NT-DUR[j]+1);t++)
            le.add(delta[j][t].mul((t+1)));
        p.newCtr("Link_"+(j+1), le.eq(start[j]));
    }

    for(j=0;j<NJ;j++)            /* Unique start time for each job */
    {
        le = new XPRBexpr();
        for(t=0;t<(NT-DUR[j]+1);t++) le.add(delta[j][t]);
        p.newCtr("One_"+(j+1), le.eq(1));
    }

    p.setObj(z);                 /* Define and set objective */

    for(j=0;j<NJ;j++) start[j].setUB(NT-DUR[j]+1);
                                /* Upper bounds on "start" var.s */
}

static void jobsSolve()
{
    int j,t,statmip;

    for(j=0;j<NJ;j++)
        for(t=0;t<NT-DUR[j]+1;t++)
            delta[j][t].setDir(XPRB.PR, 10*(t+1));
                                /* Give highest priority to var.s for earlier start times */

    p.setSense(XPRB.MINIM);
    p.solve("g");                /* Solve the problem as MIP */
    statmip = p.getMIPStat();     /* Get the MIP problem status */

    if((statmip == XPRB.MIP_SOLUTION) ||
        (statmip == XPRB.MIP_OPTIMAL))
    {
        /* An integer solution has been found */
        System.out.println("Objective: " + p.getObjVal());
                                /* Print solution for all start times */
        for(j=0;j<NJ;j++)
            System.out.println(start[j].getName() + ": " +
                                start[j].getSol());
    }
}

public static void main(String[] args)

```

```

{
    bcl = new XPRB();           /* Initialize BCL */
    p = bcl.newProb("Jobs");    /* Create a new problem */

    jobsModel();               /* Problem definition */
    jobsSolve();               /* Solve and print solution */
}
}

```

The definition of SOS is similar to the definition of constraints.

```

static XPRBsos[] set;
static XPRBprob p;

static void jobsModel()
{
    ...
    delta = new XPRBvar[NJ][NT];
    for(j=0; j<NJ; j++)          /* Variables for each job */
        for(t=0; t<(NT-DUR[j]+1); t++)
            delta[j][t] = p.newVar("delta" + (j+1) + (t+1), XPRB.PL, 0, 1);

    set = new XPRBsos[NJ];
    for(j=0; j<NJ; j++)          /* SOS definition */
    {
        le = new XPRBexpr();
        for(t=0; t<(NT-DUR[j]+1); t++)
            le.add(delta[j][t].mul((t+1)));
        set[j] = p.newSos("sosj", XPRB.S1, le);
    }
}

```

Branching directives for the SOSs are added as follows.

```

for(j=0; j<NJ; j++) set[j].setDir(XPRB.DN);
/* First branch downwards on sets */

```

Adding the following two lines during or after the problem definition will print the problem to the standard output and export the matrix to a file respectively.

```

p.print();           /* Print out the problem def. */
p.exportProb(XPRB.MPS, "expl1");
/* Output matrix to MPS file */

```

Similarly to what has been shown for the problem formulation in C and C++, we may read data from file and use index sets in the problem formulation. Only a few changes and additions to the basic model formulation are required for the creation and use of index sets. However, if we want to read in a data file in the format accepted by the C functions `XPRBreadline` and `XPRBreadarrline` (that is, using `'!'` as commentary sign, and `','` as separators, and skip blanks and empty lines), we need to configure the data file access in Java.

In the following program listing we leave out the method `jobsSolve` because it remains unchanged from the previous.

```

import java.io.*;
import com.dashoptimization.*;

public class xbexpl1i
{
    static final int MAXNJ = 4;          /* Max. number of jobs */
    static final int NT = 10;           /* Time limit */

    static int NJ = 0;                  /* Number of jobs read in */
    static final double[] DUR;          /* Durations of jobs */

    static XPRBindexSet Jobs;           /* Job names */
    static XPRBvar[] start;             /* Start times of jobs */
    static XPRBvar[][] delta;           /* Binaries for start times */
}

```



```

static XPRBvar z;                                /* Max. completion time */

static XPRB bcl;
static XPRBprob p;

    /**** Initialize the stream tokenizer ****/
static StreamTokenizer initST(FileReader file)
{
    StreamTokenizer st=null;

    st= new StreamTokenizer(file);
    st.commentChar('!');                        /* Use character '!' for comments */
    st.eolIsSignificant(true);                  /* Return end-of-line character */
    st.ordinaryChar(',');                       /* Use ',' as separator */
    st.parseNumbers();                          /* Read numbers as numbers (not strings)*/
    return st;
}

    /**** Read data from files ****/
static void readData() throws IOException
{
    FileReader datafile=null;
    StreamTokenizer st;
    int i;

                                /* Create a new index set */
    Jobs = p.newIndexSet("Jobs", MAXNJ);
    DUR = new double[MAXNJ];

    datafile = new FileReader("durations.dat");
    st = initST(datafile);
    do
    {
        do
        {
            st.nextToken();
        } while(st.ttype==st.TT_EOL); /* Skip empty lines */
        if(st.ttype != st.TT_WORD) break;
        i=Jobs.addElement(st.sval);
        if(st.nextToken() != ',') break;
        if(st.nextToken() != st.TT_NUMBER) break;
        DUR[i] = st.nval;
        NJ+=1;
    } while( st.nextToken() == st.TT_EOL && NJ<MAXNJ);
    datafile.close();
    System.out.println("Number of jobs read: " + Jobs.getSize());
}

static void jobsModel()
{
    XPRBexpr le;
    int j,t;

    start = new XPRBvar[NJ];
    for(j=0;j<NJ;j++)                /* Start time variables with bounds */
        start[j] = p.newVar("start",XPRB.PL,0,NT-DUR[j]+1);
    z = p.newVar("z",XPRB.PL,0,NT); /* Makespan variable */

    delta = new XPRBvar[NJ][NT];
    for(j=0;j<NJ;j++)                /* Binaries for each job */
        for(t=0;t<(NT-DUR[j]+1);t++)
            delta[j][t] =
                p.newVar("delta"+Jobs.getIndexName(j)+"_"+(t+1),
                    XPRB.BV);

    for(j=0;j<NJ;j++)                /* Calculate max. completion time */
        p.newCtr("Makespan", start[j].add(DUR[j]).lEq(z));

    p.newCtr("Prec", start[0].add(DUR[0]).lEq(start[2]));
                                /* Precedence rel. between jobs */

    for(j=0;j<NJ;j++)                /* Linking start times & binaries */
    {
        le = new XPRBexpr();
        for(t=0;t<(NT-DUR[j]+1);t++)

```

```

        le.add(delta[j][t].mul((t+1)));
        p.newCtr("Link_"+(j+1), le.eql(start[j]));
    }

    for(j=0;j<NJ;j++)                /* Unique start time for each job */
    {
        le = new XPRBexpr();
        for(t=0;t<(NT-DUR[j]+1);t++) le.add(delta[j][t]);
        p.newCtr("One_"+(j+1), le.eql(1));
    }

    p.setObj(z);                      /* Define and set objective */
}

public static void main(String[] args)
{
    bcl = new XPRB();                 /* Initialize BCL */
    p = bcl.newProb("Jobs");          /* Create a new problem */

    try
    {
        readData();                  /* Data input from file */
    }
    catch(IOException e)
    {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    jobsModel();                     /* Problem definition */
    jobsSolve();                     /* Solve and print solution */
}
}

```

6.1.2 QCQP Example

The following is an implementation with BCL Java of the QCQP example described in Section 3.4.1:

```

import java.io.*;
import com.dashoptimization.*;

public class xbairport
{
    static final int N = 42;
    /* Initialize the data tables:
    static final double CX[] = ...
    static final double CY[] = ...
    static final double R[] = ...
    */
    public static void main(String[] args) throws IOException
    {
        XPRB bcl;
        XPRBprob prob;
        int i,j;
        XPRBvar[] x,y;                /* x-/y-coordinates to determine */
        XPRBexpr qe;
        XPRBctr cobj, c;

        bcl = new XPRB();              /* Initialize BCL */
        prob = bcl.newProb("airport"); /* Create a new problem in BCL */

        /**** VARIABLES ****/
        x = new XPRBvar[N];
        for(i=0;i<N;i++)
            x[i] = prob.newVar("x(" + (i+1) + ")", XPRB.PL, -10, 10);
        y = new XPRBvar[N];
        for(i=0;i<N;i++)
            y[i] = prob.newVar("y(" + (i+1) + ")", XPRB.PL, -10, 10);

        /****OBJECTIVE****/
        /* Minimize the total distance between all points */
        qe = new XPRBexpr();
    }
}

```

```

        for(i=0;i<N-1;i++)
            for(j=i+1;j<N;j++) qe .add((x[i].add(x[j].mul(-1))).sqr())
                                .add((y[i].add(y[j].mul(-1))).sqr());
        cobj = prob.newCtr("TotDist", qe);
        prob.setObj(cobj);          /* Set objective function */

    /**** CONSTRAINTS ****/
    /* All points within given distance of their target location */
    for(i=0;i<N;i++)
        c = prob.newCtr("LimDist", (x[i].add(-CX[i])).sqr()
                        .add( (y[i].add(-CY[i])).sqr()) .lEq(R[i]) );

    /****SOLVING + OUTPUT****/
    prob.setSense(XPRB.MINIM);      /* Sense of optimization */
    prob.solve("");                 /* Solve the problem */

    System.out.println("Solution: " + prob.getObjVal());
    for(i=0;i<N;i++)
        System.out.println(x[i].getName() + ": " + x[i].getSol() +
                           ", " + y[i].getName() + ": " + y[i].getSol());
    }
}

```

6.1.3 Error handling

If an error occurs, BCL Java raises exceptions. A large majority of these exceptions are of class `XPRBError`, during initialization of class `XPRBlicenseError`, and if file access is involved (such as in method `exportProb`) of class `IOException`. For simplicity's sake most of the Java program examples in this manual omit the error handling. Below we show a Java implementation of the example of user error handling with BCL from Section 3.5. Other features demonstrated by this example include

- redirection of the BCL output stream for the whole program and for an individual problem;
- setting the BCL message printing level;
- forcing garbage collection for a problem.

```

import java.io.*;
import com.dashoptimization.*;

public class xbexpl3
{
    static XPRB bcl;

    /*****/

    public static void modexpl3(XPRBprob prob) throws XPRBError
    {
        XPRBvar[] x;
        XPRBexpr cobj;
        int i;

        x = new XPRBvar[3];          /* Create the variables */
        for(i=0;i<2;i++) x[i] = prob.newVar("x_"+i, XPRB.UI, 0, 100);

        /* Create the constraints:
           C1: 2x0 + 3x1 >= 41
           C2: x0 + 2x1 = 13 */
        prob.newCtr("C1", x[0].mul(2).add(x[1].mul(3)) .gEq(41));
        prob.newCtr("C2", x[0].add(x[1].mul(2)) .eq(13));

        /* Uncomment the following line to cause an error in the model that
           triggers the error handling: */

        // x[2] = prob.newVar("x_2", XPRB.UI, 10, 1);

        /* Objective: minimize x0+x1 */
        cobj = new XPRBexpr();
    }
}

```

```

    for(i=0;i<2;i++) cobj.add(x[i]);
    prob.setObj(cobj);          /* Select objective function */
    prob.setSense(XPRB.MINIM);  /* Set objective sense to minimization */

    prob.print();               /* Print current problem definition */

    prob.solve("");             /* Solve the LP */
    System.out.println("Problem status: " + prob.getProbStat() +
        "    LP status: " + prob.getLPStat() +
        "    MIP status: " + prob.getMIPStat());

/* This problem is infeasible, that means the following command will fail.
   It prints a warning if the message level is at least 2 */

    System.out.println("Objective: " + prob.getObjVal());

    for(i=0;i<2;i++)           /* Print solution values */
        System.out.print(x[i].getName() + ":" + x[i].getSol() + ", ");
    System.out.println();
}

/*****

public static void main(String[] args)
{
    FileWriter f;
    XPRBprob prob;

    try
    {
        bcl = new XPRB();          /* Initialize BCL */
    }
    catch(XPRBlicenseError e)
    {
        System.err.println("BCL error "+ e.getErrorCode() + ": " + e.getMessage());
        System.exit(1);
    }

    bcl.setMsgLevel(2);           /* Set the printing flag. Try other values:
                                   0 - no printed output,
                                   2 - print warnings, 3 - all messages */

    try
    {
        f=new FileWriter("expl3out.txt");
        bcl.setOutputStream(f);    /* Redirect all output from BCL to a file */

        prob = bcl.newProb("Expl3"); /* Create a new problem */
        prob.setOutputStream();     /* Output for this prob. on standard output */
        modexpl3(prob);            /* Formulate and solve the problem */

        prob.setOutputStream(f);    /* Redirect problem output to file */
        prob.print();              /* Write to the output file */
        f.close();

        prob=null;                /* Delete the problem */
        System.gc();               /* Force garbage collection */
        System.runFinalization();

        System.err.flush();
    }
    catch(IOException e)
    {
        System.err.println(e.getMessage());
        System.exit(1);
    }
    catch(XPRBError e)
    {
        System.err.println("BCL error "+ e.getErrorCode() + ": " + e.getMessage());
        System.exit(1);
    }
}

```

6.2 Java class reference

The complete set of classes of the BCL Java interface is summarized in the following list. For a detailed documentation of the Java interface the reader is referred to the BCL Javadoc that is part of the Xpress-MP distribution.

XPRB	Initialization and general settings, definition of all parameters.
XPRBprob	Problem definition, including methods for creating and deleting the modeling objects, problem solving, changing settings, and retrieving solution information.
XPRBvar	Methods for modifying and accessing variables.
XPRBctr	Methods for constructing, modifying and accessing constraints.
XPRBcut	Methods for constructing, modifying and accessing cuts.
XPRBsos	Methods for constructing, modifying and accessing Special Ordered Sets.
XPRBindexSet	Methods for constructing and accessing index sets and accessing set elements.
XPRBbasis	Methods for accessing bases.
XPRBexpr	Methods for constructing linear and quadratic expressions.
XPRBrelation	Methods for constructing linear or quadratic relations from expressions (extends <code>XPRBexpr</code>).
XPRBerror	Exception raised by BCL errors (extends <code>Error</code>).
XPRBlicenseError	Exception raised by BCL licensing errors (extends <code>XPRBerror</code>).
XPRBlicense	For OEM licensing.

All Java classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsos`, `XPRBindexSet`, `XPRBbasis`) take the same names, with the exception of `XPRBindexSet`. It is possible to obtain the Xpress-Optimizer problem corresponding to a BCL Java problem by using method `getXPRSProb` of class `XPRBprob`, please see Section B.7 for further detail on using BCL with the Optimizer library.

Most of the methods of the classes with direct correspondence with C modeling objects call standard BCL C functions and return their result. Where the C functions return 0 or 1 to indicate success or failure of the execution of a function the Java methods have return type `void`, raising an exception if an error occurs.

An important class that does not correspond to any standard BCL modeling object is class `XPRB` that contains methods relating to the initialization and the general status of the software and also the definition of all parameters. This means, any parameter with the prefix `XPRB_` in standard BCL is referred to as a constant of the Java class `XPRB`. For example, `XPRB_BV` in standard BCL becomes `XPRB.BV` in Java.

In Java, it is not possible to overload operators as this is the case in the C++ interface; instead, a set of simple methods is provided, for example, `add` or `eq1` that have been overloaded to accept various types and numbers of parameters. Some additional classes have been introduced to aid the termwise definition of constraints. Linear and quadratic expressions (class `XPRBexpr`) are required in the definition of constraints and Special Ordered Sets. Linear or quadratic relations (class `XPRBrelation`), may be used as an intermediary in the definition of constraints.

A few other additional classes are related to error handling and licensing, namely `XPRBerror`, `XPRBlicense`, and `XPRBlicenseError` (overloads `XPRBerror`). License errors are raised by

the initialization of BCL, all other BCL errors are handled by exceptions of the type `XPRBError`. Output functions involving file access (in particular matrix output with `exportProb`) may also generate exceptions of type `IOException`. The class `XPRBlicense` only serves for OEM licensing; for further detail please see the Xpress-MP OEM licensing documentation.

Appendix

Appendix A

BCL error messages

There are two types of error messages displayed by BCL. Those marked 'E' (for Error) in the following list stop the execution of the program. Those marked 'W' (for Warning) do not interrupt the program. The marker 'fct' indicates that the name of the function where the error occurred will be printed out.

E-1502 *Not enough memory.*

It is not possible to allocate the required amount of memory needed for BCL objects.

E-1504 *Dictionary cannot be re-initialized.*

Dictionary sizes can only be set immediately after the creation of a problem.

E-1505 (fct) *No variable given.*

Function *fct* requires a variable of type `XPRBvar` as an input parameter. Check whether the variable has been created (functions `XPRBnewvar` or `XPRBnewarrvar`).

E-1506 (fct) *No array of variables given.*

Function *fct* requires an array of variables of type `XPRBarrvar` as an input parameter. Check whether the array has been created (function `XPRBnewarrvar` or alternatively functions `XPRBstartarrvar` and `XPRBendarrvar`).

E-1507 (fct) *No constraint given.*

Function *fct* requires a constraint of type `XPRBctr` as an input parameter. Check whether the constraint has been created (functions `XPRBnewctr`, `XPRBnewsum`, `XPRBnewarrsum`, or `XPRBnewprec`).

E-1508 (fct) *No SOS given.*

Function *fct* requires a SOS of type `XPRBsos` as an input parameter. Check whether the set has been created (functions `XPRBnewsos`, `XPRBnewsosrc`, or `XPRBnewsosw`).

E-1509 (fct) *No cut given.*

Function *fct* requires a cut of type `XPRBcut` as an input parameter. Check whether the cut has been created (functions `XPRBnewcut`, `XPRBnewcutsum`, `XPRBnewcutarrsum`, or `XPRBnewcutprec`).

E-1510 (fct) *No basis given.*

Function *fct* requires a basis of type `XPRBbasis` as an input parameter. Check whether the basis has been saved (function `XPRBsavebasis`).

E-1512 (fct) No array of constants given.

Function *fct* requires an array of constants as an input parameter.

W-1513 (fct) No variable given.

Function *fct* requires a variable of type `XPRBvar` as an input parameter. The command is being ignored.

W-1514 (fct) No constraint given.

Function *fct* requires a constraint of type `XPRBctr` as an input parameter. The command is being ignored.

E-1515 (fct) Problem has no 'name'.

The problem definition is incomplete (at least one variable and one constraint or one non-zero objective coefficient must be defined).

W-1516 (fct) Problem has no 'name'.

The problem definition may be incomplete (at least one variable and one constraint or one non-zero objective coefficient must be defined).

W-1518 (fct) No SOS given.

Function *fct* requires a Special Ordered Set of type `XPRBsos` as an input parameter. The command is being ignored.

W-1519 (fct) No cut given.

Function *fct* requires a cut of type `XPRBcut` as an input parameter. The command is being ignored.

W-1520 (fct) No solution available or problem modified since last solved.

Function *fct* is trying to access solution information which is not available for the current problem.

E-1521 Xpress-Optimizer error getting objective function value.

The objective function value cannot be obtained from Xpress-Optimizer.

E-1522 Xpress-Optimizer error getting 'name' status.

Xpress-Optimizer solution status information cannot be obtained.

E-1523 Unknown solution option 'char'.

Possible options for `XPRBsolve` include 'b', 'd', 'g', 'l', 'p'. Refer to the reference manual for details.

E-1524 (fct) Xpress-Optimizer error num during 'name'.

An Xpress-Optimizer error has occurred while executing the Optimizer function *name*. Refer to the Optimizer Reference Manual for details on the error number *num*.

W-1525 (fct) *Different problem loaded in Xpress-Optimizer.*

(Solution) information is being sought from the Xpress-Optimizer on a problem that is not the active problem in Xpress-Optimizer. It may be necessary to (re)solve the problem to access this information, or at least, reload the matrix.

E-1526 (fct) *Empty problem or problem not loaded in Xpress-Optimizer.*

(Solution) information is being sought on a problem that has not yet been loaded into Xpress-Optimizer. It may be necessary to solve the problem to access this information, or at least, load the matrix into Xpress-Optimizer.

E-1530 (fct) *Inconsistent bounds for variable 'name' (bdl,bdu).*

The lower bound is greater than the upper bound for the given variable.

E-1531 (fct) *Incorrect type for variable 'name'.*

No type, or an incorrect type, has been specified for a variable. See the list of possible values in the reference manual (function `XPRBsetvartype`).

E-1535 (fct) *Incorrect type for constraint 'name'.*

No type, or an incorrect type, has been specified for a constraint. See the list of possible values in the reference manual (function `XPRBsetctrtype`).

E-1536 (fct) *Inconsistent range for constraint 'name' (bdl,bdu).*

The lower range bound is greater than the upper bound for the given constraint.

E-1540 (fct) *Trying to modify a closed array of variables ('name').*

It is not possible to make changes to an array of variables after its definition has been terminated with `XPRBendarrvar`.

E-1541 (fct) *Index num1 out of range for an array of variables ('name' max = num2).*

Trying to store too many elements in an array of variables or addressing an index beyond its size.

E-1542 (fct) *Trying to add an entry ('name') to a complete array of variables ('name').*

If the number of elements of the array of variables corresponds to its size, it is not possible to add any further variables.

E-1543 (fct) *Trying to close an incomplete array of variables ('name').*

Not all elements of an array of variables that is being closed with `XPRBendarrvar` have been defined.

E-1545 (fct) *Wrong type for SOS 'name'.*

No type, or an incorrect type, has been specified for a SOS. See the list of possible values in the reference manual (function `XPRBgetsostype`).

E-1546 (fct) *Name too long (max = num 'name').*

A user-defined name exceeds the maximum length (see documentation of function `XPRBnewname`).

E-1547 (fct) Wrong directive type.

No type, or an incorrect type, has been specified for a directive. See the list of possible values in the reference manual (functions [XPRBsetvardir](#) or [XPRBsetsosdir](#)).

E-1550 (fct) No index set given.

Function *fct* requires an index set of type `XPRBidxset` as an input parameter. Check whether the index set has been created (function [XPRBnewidxset](#)).

W-1551 (fct) No name given for an element of an index set.

Function *fct* requires an index name as input parameter. The command is being ignored.

W-1552 (fct) No index set given.

Function *fct* requires an index set of type `XPRBidxset` as an input parameter. The command is being ignored.

W-1555 Incorrect IIS index given (num).

The specified index value *num* does not correspond to an IIS set (IIS set indices are positive numbers). The command is being ignored.

E-1560 No Xpress-BCL license found. Please contact Xpress Support to obtain a license

No valid BCL license has been found. If you did install a license, check whether you have copied it to the right place and that all environment variables and paths for BCL and the Xpress-Optimizer are set correctly.

E-1561 (fct) Initialization failed (value: num).

Xpress-Optimizer could not be initialized (error code *num*).

W-1562 (fct) Working with Student License.

BCL is running in Student mode; this mode implies restrictions to the available functionality and to the accepted problem size.

E-1563 (fct) Inconsistency during matrix generation.

Internal error during the matrix generation.

E-1565 (fct) Internal error.

Internal error during the matrix generation.

E-1566 Name too long: 'name'.

A user-defined or BCL composed name exceeds the maximum length. (Remember that BCL adds indices to names if they already exist.)

E-1567 (fct) Size limits of the Student License exceeded.

The specified model is too large to be run with the Student License.

W-1568 Operation fct not allowed in Student License.

You are not authorized to execute function *fct* with the Student License of BCL.

W-1570 *XOSL: text*

Refer to the Optimizer Reference Manual for the indicated error.

E-1571 *text*

The initialization has not found a valid license.

W-1580 *Unknown output file format format.*

Refer to the documentation of function `XPRBexportprob` for admissible output format options.

E-1582 *Internal error writing MPS file.*

Please contact Xpress Support.

W-1587 *Switch to cut mode.*

The cut mode probably needs to be enabled (function `XPRBsetcutmode`) before this function is called.

E-1591 *(fct) Non-quadratic term.*

A term of the objective function has a power higher than 2.

Appendix B

Using BCL with the Optimizer library

BCL provides both modeling and basic optimization functions, which correspond to the functionality of Xpress-Mosel, or of the Xpress-Modeler and the functions of the Xpress-Optimizer library in 'Console Mode', respectively. However, if the user wishes to access the more advanced features of the Optimizer, obtain additional information, or change algorithm settings, the relevant Optimizer library functions have to be used directly.

The following sections explain in more detail how to use Optimizer library functions within a BCL program. The first section lists those functions which are compatible with BCL. It is followed by some general remarks about initialization, loading the matrix and the use of indices. The last section contains some typical examples for the use of BCL-compatible Optimizer functions in BCL programs.

Important: If a program uses Optimizer library functions the Optimizer header file has to be included in addition to the BCL header file. That is, the first lines of the program should contain the following:

```
#include "xprb.h"
#include "xprs.h"
```

B.1 Switching between libraries

Generally speaking, there are two types of Optimizer library functions: those that access information about a problem or change settings for the search algorithms, and those that make changes to the problem definition. The first group of functions may be used in a BCL program without any problem. The second group requires the user to switch completely to the Optimizer library, for instance after a problem has been defined in BCL and the matrix has been loaded into the Optimizer.

B.1.1 BCL-compatible Optimizer functions

The following Optimizer library functions may be used *with BCL* (however, some caution is required with all functions that take column or row indices as input parameters, see Section [B.4](#) below. Furthermore, the solution information in BCL is only updated automatically at the end of the search, in the global callbacks it needs to be updated by calling `XPRBsync` with the parameter `XPRB_XPRS_SOL`):

- **setting and accessing problem and control parameters:** functions `XPRSsetintcontrol`, `XPRSgetintcontrol`, `XPRSgetintattrib` etc.;
- **output and saving:** functions `XPRSsave`, `XPRSwritebasis`, `XPRSrange`, `XPRSiis`, `XPRSwriteprtsol`, `XPRSwritesol`, `XPRSwriteprtrange`, `XPRSwriterange`, `XPRSgetlpsol`, `XPRSgetmipsol`, `XPRSwriteomni`, `XPRSwriteprob`, all logging and solution callbacks with the exception of `XPRSsetcbmessage` that is used by BCL and must *not* be re-defined by the user;

- *accessing information*: all functions `XPRSget...`;
- *settings for algorithms*: `XPRSreaddir`, `XPRSloaddirs`, `XPRSreadbasis`, `XPRSloadbasis`, `XPRSloadsecurevecs`, `XPRSscale`, `XPRSftran`, `XPRsbtran`, all global callbacks;
- *cut manager*.

B.1.2 Incompatible Optimizer functions

The following Optimizer library functions may be used only *after or in place of BCL*:

- *changing, adding, and deleting matrix elements*: all functions `XPRSadd...`, `XPRSchg...`, `XPRScdel...`;
- *solution algorithms*: `XPRSminim`, `XPRSmaxim`, `XPRScglobal`;
- *input of data or problem(s)*: `XPRSreadprob`, `XPRSloadlp`, `XPRSloadglobal`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSalter`, `XPRSsetprobname`;
- *manipulation of the matrix*: `XPRSrestore`;

Once any of the functions in the preceding list have been called for a given problem, the information held in BCL may be different from the problem in the Optimizer and it is not possible to update BCL accordingly. The program must therefore continue using only Optimizer library functions on that problem, that is, switch completely to the Optimizer library. The 'switching' from BCL to the Optimizer library always refers to a single problem. If other problems are being worked on in parallel, for which none of the above incompatible function have been called, users can continue to work with them using BCL functions.

B.2 Initialization and termination

The Optimizer library is initialized at the same time as BCL and so there is no need to call the Optimizer library initialization function, `XPRSinit`, from a user program. In standard use of BCL the function `XPRBnewprob` calls the BCL initialization function `XPRBinit` that automatically initializes the Optimizer if this is the first call to `XPRBinit`. In very large applications or integration with other systems it may be preferable to call `XPRBinit` explicitly to separate the initialization from the definition of the problem(s).

At the end of the program, the normal BCL termination routine should be applied, first releasing any memory associated to problems using `XPRBdelprob` and subsequently calling `XPRBfree` to tidy up. These routines also free memory associated with the Optimizer library and hence neither of the `XPRSdestroyprob` or `XPRSfree` functions must be used. However, if one wishes to continue working with the Optimizer after terminating BCL, the Optimizer needs to be initialized (possibly before initializing BCL) and terminated separately.

Thus, the standard use of BCL is as follows:

```
XPRBprob prob;
prob = XPRBnewprob("Example1");
...                               /* Define and solve the problem */
```

Integration of a BCL problem into some larger application:

```
XPRBprob prob;
XPRBinit("");
...                               /* Perform other initialization tasks */
prob = XPRBnewprob("Example1");
...                               /* Define and solve the problem */
XPRBdelprob(prob);
XPRBfree();
```

B.3 Loading the matrix

BCL loads the matrix into the Optimizer library whenever (through BCL) an action is required from the Optimizer and the matrix in the Optimizer does not correspond to the one in BCL. This means, if a user wishes to switch to using Optimizer library commands, for instance for performing the optimization, he should explicitly load the current BCL problem into the Optimizer (function `XPRBloadmat`).

Since both BCL and the Optimizer require separate problem pointers to specify the problem being worked on, there is an issue about how to obtain the Optimizer problem pointer referring to a problem just loaded by BCL. Such issues are handled using the function `XPRBget-XPRSProb`, which returns the required Optimizer pointer. It should be noted that no call to `XPRScREATEprob` is necessary in this instance, as the problem is created by BCL at the point that it is first passed to the Optimizer.

Standard use of BCL:

```
XPRBarrvar x;
...
x = XPRBnewarrvar(prob, 10, XPRB_PL, "x", 0, 100);
...
/* (Define the rest of the problem) */
XPRBmaxim(prob, "");
/* Load matrix and maximize LP problem */
for(i=0; i<10; i++)
/* Print solution values for variables */
printf("%s: %d, ", XPRBgetvarname(prob, x[i]),
XPRBgetsol(prob, x[i]));
```

Switch to using the Optimizer library after problem input with BCL:

```
XPRBprob bcl_prob;
XPRSProb opt_prob;
XPRBarrvar x;
int i, cols, len, offset;
double *sol;
char *names;

bcl_prob = XPRBnewprob("Example1"); /* Initialize BCL (and the Optimizer
library) and create a new problem */
x = XPRBnewarrvar(bcl_prob, 10, XPRB_PL, "x", 0, 100);
...
/* Define the rest of the problem */
XPRBloadmat(bcl_prob);
/* Load matrix into the Optimizer */
opt_prob = XPRBgetXPRSProb(bcl_prob);
/* Get the Optimizer problem */
XPRSmaxim(opt_prob, "");
/* Maximize the LP problem */
XPRSgetintattrib(opt_prob, XPRS_ORIGINALCOLS, &cols);
/* Get the number of columns */
sol = malloc(cols * sizeof(double));
XPRSgetlpcol(sol, NULL, NULL, NULL);
/* Get entire primal solution */
XPRSgetnamelist(opt_prob, 2, NULL, 0, &len, 0, cols-1);
/* Get number of bytes required for
retrieving names */
names = (char *)malloc(len*sizeof(char));
XPRSgetnamelist(opt_prob, 2, names, len, NULL, 0, ncol-1);
/* Get the variable names */
offset=0;
for(i=0; i<cols; i++) {
/* Print all solution values */
printf("%s: %g, ", names+offset, sol[i]);
offset += strlen(names+offset)+1;
}
```

B.4 Indices of matrix elements

The row and column indices that are returned by the BCL functions `XPRBgetrownum` and `XPRB-getcolnum` correspond to the position of variables and constraints in the unresolved matrix with empty rows or columns removed. The position of matrix elements may be modified by

the presolve/preprocessing algorithms. That means, if these algorithms are not switched off (control parameters `XPRS_PRESOLVE` and `XPRS_MIPPRESOLVE`), the indices for variables and constraints held by BCL should not be used with any Optimizer library functions. The same rule applies to any other variable or constraint-specific information, such as solution and dual values. This problem does *not* occur within BCL (that is, if only BCL functions are used) since the solution information is accessible only after the optimization run has finished and the postsolve has been performed by the Optimizer.

An exception from the rule stated above are the Optimizer library functions `XPRSgetlpsol` / `XPRSgetmipsol`: `XPRSgetlpsol` may be used, for instance, in Optimizer library callback functions during the global search to access the current solution values, and in combination with the indices for variables and constraints held by BCL. This is possible because `XPRSgetlpsol` / `XPRSgetmipsol` return the postsolved solution (depending on the setting of the control parameter `XPRS_SOLUTIONFILE`).

B.5 Using BCL-compatible functions

The Optimizer library functions that are most likely to be used in a BCL program are those for setting and accessing control and problem parameters, as shown in the following examples. The control parameters can be set and accessed at any time after the software has been initialized (see Section B.2). The problem attributes only return the problem-specific values once the problem has been loaded into the Optimizer. Note that all the parameters take their default values at the beginning of a BCL program but they are *not* reset if several problems are solved in a single program and changes are made to the parameter values along the way.

Setting control parameters:

```
XPRBprob bcl_prob;
XPRSprob opt_prob;

bcl_prob = XPRBnewprob("Example1"); /* Initialize BCL (and the Optimizer
                                   library) and create a new problem */
...                               /* Define the problem */
XPRBloadmat(bcl_prob);
opt_prob = XPRBgetXPRSprob(bcl_prob);
XPRSsetintcontrol(opt_prob,XPRS_MAXTIME, 60);
                                   /* Set a time limit of 60 seconds */
XPRSsetdblcontrol(opt_prob,XPRS_MIPADDCUTOFF, 0.999);
                                   /* Set an ADDCUTOFF value */
XPRBmaxim(bcl_prob,"");           /* Load matrix and maximize LP problem */
```

Accessing problem parameters:

```
int rows;
XPRBprob bcl_prob;
XPRSprob opt_prob;

bcl_prob = XPRBnewprob("Example1"); /* Initialize BCL (and the Optimizer
                                   library) and create a new problem */
...                               /* Define the problem */
XPRBloadmat(bcl_prob);           /* Load matrix into the Optimizer */
opt_prob = XPRBgetXPRSprob(bcl_prob);
XPRSgetintattrib(opt_prob, XPRS_ORIGINALROWS, &rows);
                                   /* Get number of rows */
XPRBmaxim(bcl_prob,"");           /* Maximize the LP problem */
```

Another likely set of functions are the Optimizer library callbacks for solution printout and possibly for directing the branch and bound search (see the remarks about indices in Section B.4):

```
void XPRS_CC printsol(XPRSprob opt_prob,void *my_object)
{
    XPRBprob bcl_prob
    XPRBvar x;
```



```

int num;

bcl_prob = (XPRBprob)my_object;
XPRSgetintattrib(opt_prob, XPRS_INTSOL, &num);
/* Get number of the solution */
XPRSsetintcontrol(opt_prob, XPRS_SOLUTIONFILE, 0);
XPRBsync(bcl_prob, XPRB_XPRS_SOL);
/* Update BCL solution values */
XPRBprintf(bcl_prob, "Solution %d: Objective value: %g\n",
            num, XPRBgetobjval(bcl_prob));
x = XPRBgetbyname(bcl_prob, "x_1", XPRB_VAR);
if(XPRBgetcolnum(x)>-1) /* Test whether variable is in the
                        matrix */
    XPRBprintf(bcl_prob, "%s: %g\n", XPRBgetvarname(x), XPRBgetsol(x));
XPRSsetintcontrol(opt_prob, XPRS_SOLUTIONFILE, 1);
}

int main(int argc, char **argv)
{
    XPRBprob bcl_prob;
    XPRSprob opt_prob;
    XPRBvar x;

    bcl_prob = XPRBnewprob("Example1"); /* Initialize BCL (and the Optimizer
                                        library) and create a new problem */
    x = XPRBnewvar(bcl_prob, XPRB_BV, "x_1", 0, 1); /* Define a variable */
    ... /* Define the rest of the problem */
    opt_prob = (XPRSprob)XPRBgetXPRSprob(bcl_prob);
    XPRSsetcbintsol(opt_prob, printsol, bcl_prob);
    /* Define an integer solution callback */
    XPRBmaxim(bcl_prob, "g"); /* Maximize the MIP problem */
}

```

B.6 Using the Optimizer with BCL C++

Everything that has been said above about the combination of BCL and Xpress-Optimizer functions remains true if the BCL program is written in C++.

The examples of BCL-compatible Optimizer functions in the previous section become:

Setting and accessing parameters:

```

int rows;
XPRSprob opt_prob;
XPRBprob bcl_prob("Example1"); // Initialize BCL (and the Optimizer
                                // library) and create a new problem
...                             // Define the problem
bcl_prob.loadMat();
opt_prob = bcl_prob.getXPRSprob();
XPRSsetintcontrol(opt_prob, XPRS_MAXTIME, 60);
/* Set a time limit of 60 seconds
XPRSsetdblcontrol(opt_prob, XPRS_MIPADDCUTOFF, 0.999);
// Set an ADDCUTOFF value
XPRSgetintattrib(opt_prob, XPRS_ORIGINALROWS, &rows);
// Get number of rows
bcl_prob.maxim(""); /* Maximize the LP problem

```

Using Xpress-Optimizer callbacks:

```

void XPRS_CC printsol(XPRSprob opt_prob, void *my_object)
{
    XPRBprob *bcl_prob
    XPRBvar x;
    int num;

    bcl_prob = (XPRBprob*)my_object;
    XPRSgetintattrib(opt_prob, XPRS_MIPSOLS, &num);
    /* Get number of the solution
    XPRSsetintcontrol(opt_prob, XPRS_SOLUTIONFILE, 0);
    bcl_prob->sync(XPRB_XPRS_SOL);

```

```

// Update BCL solution values
cout << "Solution " << num << ": Objective value: ";
cout << bprob->getObjVal() << endl;
x = bcl_prob->getVarByName("x_1");
if(x.getColNum() > -1) // Test whether variable is in the
// matrix
    cout << x.getName() << ": " << x.getSol() << endl;
XPRSsetintcontrol(opt_prob, XPRS_SOLUTIONFILE, 1);
}

int main(int argc, char **argv)
{
    XPRBprob bcl_prob;
    XPRSprb opt_prob;
    XPRBvar x;

    bcl_prob = XPRBnewprob("Example1"); // Initialize BCL (and the Optimizer
// library) and create a new problem
    x = bcl_prob.newVar("x_1", XPRB_BV); // Define a variable
    ... // Define the rest of the problem
    opt_prob = bcl_prob.getXPRSprb();
    XPRSsetcbintsol(opt_prob, printsol, &bcl_prob);
// Define an integer solution callback
    bcl_prob.maxim("g"); // Maximize the MIP problem
}

```

As in the C case, it is possible within a BCL program written in C++ to switch entirely to Xpress-Optimizer (see Section B.3).

B.7 Using the Optimizer with BCL Java

Starting with Release 3.0 of BCL it is possible to combine BCL Java problem definition with direct access to the Optimizer problem in Java. All that is said in the previous sections about BCL-compatible functions remains true. The only noticeable difference is that the Optimizer Java needs to be initialized explicitly (by calling `XPRSinit`) before the Optimizer problem is accessed.

The following are Java implementations of the code extracts showing the use of BCL-compatible functions:

Setting and accessing parameters (this code throws the exceptions `XPRSprbException` and `XPRSexception`):

```

int rows;
XPRB bcl;
XPRSprb opt_prob;
XPRBprob bcl_prob;

bcl = new XPRB(); // Initialize BCL */
bcl_prob = bcl.newProb("Example1"); // Create a new problem in BCL */
XPRS.init(); // Initialize Xpress-Optimizer */
... // Define the problem */
bcl_prob.loadMat();
opt_prob = bcl_prob.getXPRSprb();
opt_prob.setIntControl(XPRS.MAXTIME, 60);
// Set a time limit of 60 seconds */
opt_prob.setDblControl(XPRS.MIPADDCUTOFF, 0.999);
// Set an ADDCUTOFF value */
rows = opt_prob.getIntAttrib(XPRS.ORIGINALROWS);
// Get number of rows */
bcl_prob.maxim(""); // Maximize the LP problem */

```

Using Xpress-Optimizer callbacks:

```

static class IntSolCallback implements XPRSintSolListener
{
    public void XPRSintSolEvent(XPRSprb opt_prob, Object my_object)

```

```

{
    XPRBprob bcl_prob;
    XPRBvar x;
    int num;

    bcl_prob = (XPRBprob)my_object;
    try {
        num = oprob.getIntAttrib(XPRS.MIPSOLS);
        /* Get number of the solution */
        oprob.setIntControl(XPRS.SOLUTIONFILE, 0);
        bcl_prob.sync(XPRB.XPRS_SOL);
        /* Update BCL solution values */
        System.out.println("Solution " + num + ": Objective value: " +
            bcl_prob.getObjVal());
        x = bcl_prob.getVarByName("x_1");
        if(x.getColNum() > -1) /* Test whether variable is in the
            matrix */
            System.out.println(x.getName() + ": " + x.getSol());
        oprob.setIntControl(XPRS.SOLUTIONFILE, 1);
    }
    catch(XPRSprobException e) {
        System.out.println("Error " + e.getCode() + ": " + e.getMessage());
    }
}

}

public static void main(String[] args) throws XPRSexception
{
    XPRB bcl;
    XPRBprob bcl_prob;
    XPRSprob opt_prob;
    IntSolCallback cb;
    XPRBvar x;

    bcl = new XPRB(); /* Initialize BCL */
    bcl_prob = bcl.newProb("Example1"); /* Create a new problem in BCL */
    XPRS.init(); /* Initialize Xpress-Optimizer */

    x = bcl_prob.newVar("x_1", XPRB_BV); /* Define a variable */
    ... /* Define the rest of the problem */
    opt_prob = bcl_prob.getXPRSprob();
    cb = new IntSolCallback();
    opt_prob.addIntSolListener(cb, bcl_prob);
    /* Define an integer solution callback */
    bcl_prob.maxim("g"); /* Maximize the MIP problem */
}

```

Appendix C

Working with cuts in BCL

This chapter describes an extension to BCL that enables the user to define cuts in a similar way to constraints. Although cuts are just additional constraints, they are treated differently by BCL. To start with, they are defined as a separate type (`XPRBcut` instead of `XPRBctr`). Besides the type, the following differences between the representation and use of constraints and cuts in BCL may be observed:

- Cuts cannot be non-binding or ranged.
- Cuts are not stored with the problem, this is up to the user.
- Cuts have no names, but they have got an integer indicating their classification or identification number.
- Function `XPRBdelcut` deletes the cut definition in BCL, but does not influence the problem in Xpress-Optimizer if the cut has already been added to it.
- Cuts are added to the problem while it is being solved without having to regenerate the matrix; they can only be added to the matrix (using function `XPRBaddcuts`) in one of the callback functions of the Xpress-Optimizer cut manager (see the Xpress-Optimizer manual). Furthermore, they can only be defined on variables that are already contained in the matrix.

The following functions are available in BCL for handling cuts:

<code>XPRBaddcutarrterm</code>	Add multiple linear terms to a cut.	p. 27
<code>XPRBaddcuts</code>	Add cuts to a problem.	p. 28
<code>XPRBaddcutterm</code>	Add a term to a cut.	p. 29
<code>XPRBdelcut</code>	Delete a cut definition.	p. 43
<code>XPRBdelcutterm</code>	Delete a term from a cut.	p. 44
<code>XPRBgetcutid</code>	Get the classification or identification number of a cut.	p. 63
<code>XPRBgetcutrhs</code>	Get the RHS value of a cut.	p. 64
<code>XPRBgetcuttype</code>	Get the type of a cut.	p. 65
<code>XPRBnewcut</code>	Create a new cut.	p. 104
<code>XPRBnewcutarrsum</code>	Create a sum cut with individual coefficients (i c_i x_i).	p. 105
<code>XPRBnewcutprec</code>	Create a precedence cut ($v_1 + dur \ v_2$).	p. 106
<code>XPRBnewcutsum</code>	Create a sum cut (i x_i).	p. 107
<code>XPRBprintcut</code>	Print out a cut.	p. 119

<code>XPRBsetcutid</code>	Set the classification or identification number of a cut.	p. 133
<code>XPRBsetcutmode</code>	Set the cut mode.	p. 134
<code>XPRBsetcutterm</code>	Set a cut term.	p. 135
<code>XPRBsetcuttype</code>	Set the type of a cut.	p. 136

C.1 Example

The following example shows how the Xpress-Optimizer node cut manager callback may be defined to add cuts during the branch and bound search. Function `XPRBaddcuts` that adds the cuts to the problem in Xpress-Optimizer may only be called from one of the cut manager callback functions. Nevertheless, cuts may be defined at any place in the program after BCL has been initialized and the relevant variables have been defined. In order to keep the present example simple, we only create and add cuts at a single node, they are therefore created in the cut manager callback immediately before they are added to the problem. More realistically, cuts may be generated subject to a certain search tree depth or depending on the solution values of certain variables in the current LP-relaxation.

```
#include <stdio.h>
#include "xprb.h"
#include "xprs.h"

...
XPRBvar start[4];

int XPRS_CC usrcme(XPRSprob oprob, void* vd)
{
    XPRBcut ca[2];
    int num;
    int i=0;
    XPRBprob bprob;

    bprob = (XPRBprob)vd; /* Get the BCL problem */
    XPRSgetintattrib(oprob, XPRS_NODES, &num);
    if(num == 2) /* Only generate cuts at node 2 */
    {
        /* ca0: s_1+2 <= s_0 */
        ca[0] = XPRBnewcutprec(bprob, start[1], 2, start[0], 2);
        ca[1] = XPRBnewcut(bprob, XPRB_L, 2); /* ca1: 4*s_2 - 5.3*s_3 <= -17 */
        XPRBaddcutterm(ca[1], start[2], 4);
        XPRBaddcutterm(ca[1], start[3], -5.3);
        XPRBaddcutterm(ca[1], NULL, -17);
        printf("Adding constraints:\n");
        for(i=0;i<2;i++) XPRBprintcut(ca[i]);
        if(XPRBaddcuts(bprob, ca, 2)) printf("Problem with adding cuts.\n");
    }
    return 0; /* Call this func. once per node */
}

int main(int argc, char **argv)
{
    XPRBprob prob;
    XPRSprob oprob;

    prob=XPRBnewprob("CutExpl"); /* Initialization */
    for(j=0;j<4;j++) start[j] = XPRBnewvar(prob, XPRB_PL, "start", 0, 500);
    ... /* Define constraints and an objective function */
    XPRBsetcutmode(prob, 1); /* Enable the cut mode */
    oprob = XPRBgetXPRSprob(prob); /* Get the Optimizer problem */
    XPRSsetcbcutmgr(oprob, usrcme, prob); /* Def. the cut manager callback */
    XPRBsolve(prob, "g"); /* Solve the MIP problem */
    ... /* Solution output */
    return 0;
}
```

C.2 C++ version of the example

With BCL C++, the implementation of the cut example is similar to what we have seen in the previous section since the same Xpress-Optimizer functions are used.

```
#include <iostream>
#include "xprb_cpp.h"
#include "xprs.h"

using namespace std;
using namespace ::dashoptimization;

...
XPRBvar start[NJ];
XPRBprob p("Jobs"); // Initialize BCL and a new problem

int XPRS_CC usrcme(XPRSprob oprob, void* vd)
{
    XPRBcut ca[2];
    int num;
    int i=0;
    XPRBprob *bprob;

    bprob = (XPRBprob*)vd; // Get the BCL problem
    XPRSgetintattrib(oprob, XPRS_NODES, &num);
    if(num == 2) // Only generate cuts at node 2
    {
        ca[0] = bprob->newCut(start[1]+2 <= start[0], 2);
        ca[1] = bprob->newCut(4*start[2] - 5.3*start[3] <= -17, 2);
        cout << "Adding constraints:" << endl;
        for(i=0;i<2;i++) ca[i].print();
        if(bprob->addCuts(ca,2)) cout << "Problem with adding cuts." << endl;
    }
    return 0; // Call this function once per node
}

int main(int argc, char **argv)
{
    XPRSprob oprob;

    for(j=0;j<4;j++) start[j] = p.newVar("start");
    ... // Define constraints and an objective function
    oprob = p.getXPRSprob(); // Get Optimizer problem
    p.setCutMode(1); // Enable the cut mode
    XPRSsetcbcutmgr(oprob,usrcme,&p); // Def. the cut manager callback
    p.solve("g"); // Solve the problem as MIP
    ... // Solution output
    return 0;
}
```

C.3 Java version of the example

As is explained in Section B.7, before accessing directly the problem held in Xpress-Optimizer we need to initialize explicitly the Optimizer Java. The cut manager callback is implemented in Java by the class 'cutMgrListener'.

```
import java.io.*;
import com.dashoptimization.*;

public class xbcutex
{
    ...
    static XPRBvar[] start;
    static XPRB bcl;

    static class CutMgrCallback implements XPRScutMgrListener
    {
        public int XPRScutMgrEvent(XPRSprob oprob, Object data)
```

```

{
    XPRBprob bprob;
    XPRBcut[] ca;
    int num,i;

    bprob = (XPRBprob)data;          /* Get the BCL problem */

    try
    {
        num = oprob.getIntAttrib(XPRS.NODES);
        if(num == 2)                  /* Only generate cuts at node 2 */
        {
            ca = new XPRBcut[2];
            ca[0] = bprob.newCut(start[1].add(2) .lEq(start[0]), 2);
            ca[1] = bprob.newCut(start[2].mul(4) .add(start[3].mul(-5.3)) .lEq(-17), 2);
            System.out.println("Adding constraints:");
            for(i=0;i<2;i++) ca[i].print();
            bprob.addCuts(ca);
        }
    }
    catch(XPRSprobException e)
    {
        System.out.println("Error " + e.getCode() + ": " + e.getMessage());
    }
    return 0;                          /* Call this method once per node */
}

public static void main(String[] args) throws XPRSexception
{
    XPRBprob p;
    XPRSprob oprob;
    CutMgrCallback cb;

    bcl = new XPRB();                  /* Initialize BCL */
    p = bcl.newProb("Jobs");          /* Create a new problem */
    XPRS.init();                      /* Initialize Xpress-Optimizer */

    start = new XPRBvar[4];           /* Create 'start' variables */
    for(j=0;j<4;j++) start[j] = p.newVar("start");
    ...                               /* Define constraints and an objective function */

    oprob = p.getXPRSprob();           /* Get Optimizer problem */
    p.setCutMode(1);                  /* Enable the cut mode */
    cb = new CutMgrCallback();
    oprob.addCutMgrListener(cb, p);    /* Def. the cut manager callback */
    p.solve("g");                     /* Solve the problem as MIP */
    ...                               /* Solution output */
}
}

```

Index

Symbols

*****, 186
+, 186
+=, 171, 180, 186, 191, 217
-, 186
-=, 171, 180, 186
<=, 216
=, 171, 180, 186, 217
==, 216
>=, 216
[], 191

A

activity value, 10, 54, 172
add
 index element, 30, 192
array, 11, 156
 add entry, 130
 append, 35
 create, 6
 delete, 40
 incremental definition, 6
 name, 7, 55
 print, 18
 size, 7, 56
 terminate, 50

B

basis
 class, 168, 236
 delete, 10, 41, 168
 load, 10, 97, 204
 save, 10, 129, 210
 validity check, 168
BCL
 version number, 6, 94, 165
bound
 fix, 6, 53, 222
 get, 6, 57, 223, 226
 integer limit, 6, 72, 141, 223, 227
 lower, 6, 140, 227
 semi-continuous limit, 6, 72, 141, 223, 227
 upper, 6, 151, 228
branching directive
 SOS, 17, 149, 220
 variable, 6, 152, 227

C

callback
 error messages, 5, 21, 38
 messages, 5, 21, 39
 printing, 5, 21, 39
change
 constraint type, 8

 variable type, 6, 154, 228
column order, 131, 166, 211
constraint
 activity, 10, 54, 172
 add array, 7, 26
 add linear expression, 171
 add quadratic term, 31
 add term, 7, 34, 172
 add terms, 26
 change type, 8, 132, 179
 class, 170, 236
 creation, 7, 103, 206
 definition, 7, 103, 206
 delete, 8, 42, 177, 198
 delete coefficient, 8
 delete quadratic term, 46
 delete term, 49, 172
 dual, 10, 66, 173
 finding, 200
 get range, 8, 80, 174, 175
 get RHS, 8, 82, 175
 get type, 8, 62, 176
 index, 8, 83, 176
 model cut, 8, 75, 177
 name, 8, 60, 174
 number, 83, 176
 precedence, 110
 print, 18, 118, 177
 ranging information, 61, 175
 set coefficient, 7
 set model cut, 8, 142, 178
 set quadratic term, 145
 set range, 8, 146, 178
 set term, 150, 179
 set type, 132, 179
 slack, 10, 85, 176
 sum, 115
 sum with coefficients, 101
 validity check, 177
copy
 expression, 187
create
 index set, 108, 161, 207, 231
cut
 add array, 27
 add linear expression, 181
 add term, 29, 182
 add terms, 27
 class, 180, 236
 classification, 63, 182
 creation, 104, 207
 definition, 104, 207
 delete, 43, 183, 198
 delete term, 44, 182

- get RHS, 64, 183
- get type, 65, 183
- identification, 63, 182
- model, 8, 75, 142, 177, 178
- precedence, 106
- print, 119, 183
- set classification, 133, 184
- set identification, 133, 184
- set term, 135, 184
- set type, 136, 185
- sum, 107
- sum with coefficients, 105
- switch mode, 134, 211
- validity check, 183
- cut mode, 134, 211
- cuts
 - add, 28, 197
- D**
- data
 - input, 126, 127, 161, 231
 - reading, 126, 127
- data input, 14
- decimal sign
 - change, 5, 137
- delete
 - array, 40
 - basis, 168
 - constraint, 8, 42, 177, 198
 - constraint coefficient, 8
 - constraint term, 49, 172
 - cut, 43, 183, 198
 - expression term, 188
 - index set, 194
 - problem, 4, 45
 - set element, 17
 - set member, 48, 219
 - SOS, 17, 47, 199
- dictionary
 - size, 138, 211
- directive
 - delete, 36, 198
 - SOS, 17, 149, 161, 220, 231
 - variable, 6, 13, 152, 227
- directives
 - file, 158, 215
- dual values, 10, 66, 173

E

- E-1502, 239
- E-1504, 239
- E-1505, 239
- E-1506, 239
- E-1507, 239
- E-1508, 239
- E-1509, 239
- E-1510, 239
- E-1512, 240
- E-1515, 240
- E-1521, 240
- E-1522, 240
- E-1523, 240
- E-1524, 240
- E-1526, 241

- E-1530, 241
- E-1531, 241
- E-1535, 241
- E-1536, 241
- E-1540, 241
- E-1541, 241
- E-1542, 241
- E-1543, 241
- E-1545, 241
- E-1546, 241
- E-1547, 242
- E-1550, 242
- E-1560, 242
- E-1561, 242
- E-1563, 242
- E-1565, 242
- E-1566, 242
- E-1567, 242
- E-1571, 243
- E-1582, 243
- E-1591, 243
- error
 - exit, 5, 139
- error callback, 5, 21, 38
- error handling, 5, 21, 139
- error message, 5, 21, 38
- expression
 - add expression, 187
 - class, 186, 236
 - constant multiplication, 188
 - copy, 187
 - delete term, 188
 - evaluation, 188
 - linear, 164, 236
 - multiplication, 188
 - negation, 189
 - quadratic, 164, 236
 - set term, 189

F

- file
 - reading, 14
- find by name, 58
 - constraint, 200
 - index set, 200
 - SOS, 203
 - variable, 203
- format
 - real numbers, 147, 167, 213

G

- garbage collection, 234
- generate matrix, 98, 204
- getCRef, 164
- getCtrByName, 164
- getVarByName, 164
- getXPRsprob, 236

I

- IIS, see irreducible infeasible set
- incremental definition
 - array, 6
- index
 - constraint, 8, 83, 176

- variable, 6, 59, 222
- index set, 14
 - add element, 14, 30, 192
 - class, 191, 236
 - create, 14, 161, 231
 - creation, 108, 207
 - delete, 194
 - element name, 14, 68, 193
 - find element, 14, 67, 192
 - finding, 200
 - index number, 14, 67, 192
 - name, 14, 69, 193
 - print, 18, 121, 194
 - size, 14, 70, 193
 - validity check, 193
- initialization, 4, 96, 111, 197, 242, 243
 - explicit, 166
- input
 - decimal sign, 5, 137
 - file, 126, 127, 161, 231
- input file, 14
- interface pointer, 90, 153
 - callback, 37
- irreducible infeasible set
 - constraints, 71
 - number, 76, 201
 - variables, 71
- isValid, 164

L

- license, 4, 96, 242, 243
- linear expression, see expression
- linear relation, see relation
- load matrix, 5

M

- matrix
 - add cuts, 28, 197
 - column ordering, 131, 166, 211
 - generation, 98, 204
 - loading, 5
 - output, 18, 51, 161, 199, 231
- matrix generation
 - column order, 131, 166, 211
- maximize, 9, 99, 155, 205, 214
- message level, 5, 143, 167, 212, 234
- minimize, 9, 100, 155, 205, 214
- MIPPRESOLVE, 247
- MIQP, see Mixed Integer Quadratic Programming
- Mixed Integer Quadratic Programming, 19
- model cut, 8, 142, 178
- modeling object
 - finding, 58

N

- name
 - array, 7, 55
 - composing of, 109
 - constraint, 8, 60, 174
 - dictionary, 138, 211
 - index set, 14, 69, 193
 - index set element, 14, 68, 193
 - problem, 78, 201
 - SOS, 17, 87, 219

- variable, 6, 91, 223
- namespace, 159
- negation
 - expression, 189
- number
 - IIS, 76, 201

O

- objective
 - get sense, 9, 84, 202
 - quadratic, 19
 - set sense, 9, 148, 213
 - value, 10, 77, 202
- objective function, 144, 212
 - print, 122, 209
- optimize, 9, 155, 214
- Optimizer problem, 95, 203
- output
 - file, 18, 51, 199
 - redirection, 234
- output level, 5, 143, 167, 212, 234
- output stream, 234

P

- package, 229
- partial integer
 - get limit, 6, 72, 223
 - set limit, 6, 141, 227
- precedence constraint, 8, 110
- precedence cut, 106
- PRESOLVE, 247
- print
 - array, 18, 117
 - constraint, 18, 118, 177
 - cut, 119, 183
 - index set, 18, 121, 194
 - objective function, 122, 209
 - problem, 18, 123, 161, 209, 231
 - program output, 120
 - SOS, 18, 124, 220
 - text, 120
 - variable, 18, 125, 226
- print flag, 5, 143, 167, 212
- printing
 - decimal sign, 5, 137
- printing callback, 5, 21, 39
- priority, 13
 - delete, 36, 198
 - SOS, 17, 149, 220
 - variable, 6, 152, 227
- problem
 - add cuts, 28, 197
 - class, 195, 236
 - delete, 4, 45
 - delete basis, 10, 41
 - file output, 18, 51, 199
 - initialization, 4, 111, 197
 - load basis, 10, 97, 204
 - LP status, 10, 73, 201
 - maximize, 99, 205
 - minimize, 100, 205
 - MIP status, 10, 74, 201
 - objective value, 10, 77, 202
 - output, 4, 123, 209

- print, 18, 161, 231
- reset, 128, 210
- save basis, 10, 129, 210
- solve, 9, 155, 214
- status, 10, 79, 202
- synchronization, 157, 215
- program output
 - print, 18, 120

Q

- QCQP, see Quadratically Constrained Quadratic Programming
- QP, see Quadratic Programming
- quadratic expression, see expression
- Quadratic Programming, 19
- quadratic term, 31, 145
 - delete, 46
- Quadratically Constrained Quadratic Programming, 20

R

- range, 8, 80, 146, 174, 175, 178
 - get values, 8
- ranging information
 - constraint, 61, 175
 - variable, 92, 224
- read
 - data line, 126, 127
- reduced cost value, 10, 81, 224
- reference constraint, 17, 113
- relation, 164, 236
 - class, 216, 236
 - get type, 216
 - linear, 164
 - quadratic, 164
- reset
 - problem, 128, 210
- RHS, 8, 64, 82, 175, 183
- running time, 89, 165

S

- scheduling, 10
- security system, 4
- semi-continuous
 - get limit, 6, 72, 223
 - set limit, 6, 141, 227
- semi-continuous integer
 - get limit, 6, 72, 223
 - set limit, 6, 141, 227
- sense
 - objective function, 9, 84, 148, 202, 213
- set
 - cut mode, 134, 211
 - index, 14
- size
 - array, 7, 56
 - dictionary, 138, 211
 - index set, 14, 70, 193
- size limit, 242
- slack values, 10, 85, 176
- solution, 10, 86, 225
 - objective, 10, 77, 202
- solution value
 - expression, 188

- solve, 9, 99, 100, 155, 205, 214

SOS

- add array, 17, 32
- add element, 17
- add linear expression, 218
- add member, 33, 218
- add members, 32
- class, 217, 236
- creation, 16, 112–114, 161, 208, 231
- delete, 17, 47, 199
- delete element, 17
- delete member, 48, 219
- finding, 203
- name, 17, 87, 219
- print, 18, 124, 220
- set directive, 17, 149, 220
- type, 17, 88, 219
- validity check, 220
- weights, 114

- sqr, 190

- square, 190

status

- LP, 10, 73, 201
- MIP, 10, 74, 201
- problem, 10, 79, 202

- Student Edition, 119

- Student mode, 242

- student version, 20, 51, 118, 121–124, 177, 183, 194, 199, 209, 220, 226

- sum constraint, 8, 101, 115

- sum cut, 107

T

table

- sparse, 14

- time measure, 89, 165

type

- constraint, 8, 62, 132, 176, 179
- cut, 65, 136, 183, 185
- relation, 216
- SOS, 17, 88, 219
- variable, 6, 93, 154, 225, 228

V

variable

- array, 156
- array of, 6, 102
- change type, 6, 154, 228
- class, 221, 236
- creation, 6, 116, 209
- deletion callback, 37
- finding, 203
- fix value, 6, 53, 222
- get bounds, 6, 57
- get limit, 6, 72, 223
- get lower bound, 223
- get type, 6, 93, 225
- get upper bound, 226
- index, 6, 59, 222
- interface pointer, 90, 153
- lower bound, 6, 140, 227
- name, 6, 91, 223
- number, 59, 222
- print, 18, 125, 226

- print array, 117
- ranging information, 92, 224
- reduced cost, 10, 81, 224
- set directive, 6, 152, 227
- set limit, 6, 141, 227
- set type, 154, 228
- solution, 10, 86, 225
- upper bound, 6, 151, 228
- validity check, 226
- variable types, 1
- version number, 6, 94, 165

W

- W-1513, 240
- W-1514, 240
- W-1516, 240
- W-1518, 240
- W-1519, 240
- W-1520, 240
- W-1525, 241
- W-1551, 242
- W-1552, 242
- W-1555, 242
- W-1562, 242
- W-1568, 242
- W-1570, 243
- W-1580, 243
- W-1587, 243

X

- xbcut, 104
- XPRB, 165
- XPRB.getTime, 165
- XPRB.getVersion, 165
- XPRB.init, 166
- XPRB.setColOrder, 166
- XPRB.setMsgLevel, 166
- XPRB.setRealFmt, 167
- XPRB_ARR, 58
- XPRB_BV, 93, 102, 116, 154, 208, 225, 228, 236
- XPRB_CTR, 58
- XPRB_DICT_IDX, 138, 211
- XPRB_DICT_NAMES, 138, 211
- XPRB_DIR, 79, 202
- XPRB_DN, 149, 152, 220, 226
- XPRB_E, 62, 65, 101, 103–105, 107, 115, 132, 136, 176, 179, 183, 185, 216
- XPRB_ERR, 38
- XPRB_FGETS, 126, 127
- XPRB_G, 62, 65, 101, 103–105, 107, 115, 132, 136, 176, 179, 183, 185, 216
- XPRB_GEN, 79, 202
- XPRB_IDX, 58
- XPRB_I, 62, 65, 101, 103–105, 107, 115, 132, 136, 176, 179, 183, 185, 216
- XPRB_LCOST, 92, 224
- XPRB_LOACT, 61, 92, 175, 224
- XPRB_LP, 51, 199
- XPRB_LP_CUTOFF, 73, 200
- XPRB_LP_CUTOFF_IN_DUAL, 73, 201
- XPRB_LP_INFEAS, 73, 200
- XPRB_LP_OPTIMAL, 73, 200
- XPRB_LP_UNBOUNDED, 73, 201
- XPRB_LP_UNFINISHED, 73, 201

- XPRB_LP_UNSOLVED, 73, 201
- XPRB_MAXIM, 84, 148, 202, 213
- XPRB_MINIM, 84, 148, 202, 213
- XPRB_MIP_INFEAS, 74, 201
- XPRB_MIP_LP_NOT_OPTIMAL, 74, 201
- XPRB_MIP_LP_OPTIMAL, 74, 201
- XPRB_MIP_NO_SOL_FOUND, 74, 201
- XPRB_MIP_NOT_LOADED, 74, 201
- XPRB_MIP_OPTIMAL, 74, 201
- XPRB_MIP_SOLUTION, 74, 201
- XPRB_MOD, 79, 202
- XPRB_MPS, 51, 199
- XPRB_N, 62, 101, 103, 115, 132, 176, 179, 216
- XPRB_PD, 149, 152, 220, 226
- XPRB_PI, 93, 102, 116, 154, 208, 225, 228
- XPRB_PL, 93, 102, 116, 154, 208, 225, 228
- XPRB_PR, 149, 152, 220, 226
- XPRB_PU, 149, 152, 220, 226
- XPRB_R, 62, 176
- XPRB_S1, 88, 112–114, 208, 219
- XPRB_S2, 88, 112–114, 208, 219
- XPRB_SC, 93, 102, 116, 154, 208, 225, 228
- XPRB_SI, 93, 102, 116, 154, 208, 225, 228
- XPRB_SOL, 79, 202
- XPRB_SOS, 58
- XPRB_UCOST, 92, 224
- XPRB_UDN, 61, 92, 175, 224
- XPRB_UI, 93, 102, 116, 154, 208, 225, 228
- XPRB_UP, 149, 152, 220, 226
- XPRB_UPACT, 61, 92, 175, 224
- XPRB_UUP, 61, 92, 175, 224
- XPRB_VAR, 58
- XPRB_WAR, 38
- XPRB_XPRS_PROB, 157, 214
- XPRB_XPRS_SOL, 157, 214
- XPRBaddarrterm, 26
- XPRBaddcutarrterm, 27
- XPRBaddcuts, 28
- XPRBaddcutterm, 29
- XPRBaddidxel, 30
- XPRBaddqterm, 20, 31
- XPRBaddsosarrel, 32
- XPRBaddsosel, 33
- XPRBaddterm, 34
- XPRBapparrvarel, 35
- XPRBarrvar, 25
- XPRBbasis, 25, 168
- XPRBbasis.getCRef, 168
- XPRBbasis.isValid, 168
- XPRBbasis.reset, 168
- XPRBcleardir, 36
- XPRBctr, 25, 170, 171, 251
- XPRBctr.add, 171
- XPRBctr.addTerm, 172
- XPRBctr.delTerm, 172
- XPRBctr.getAct, 172
- XPRBctr.getCRef, 173
- XPRBctr.getDual, 173
- XPRBctr.getName, 174
- XPRBctr.getRange, 174
- XPRBctr.getRangeL, 174
- XPRBctr.getRangeU, 174
- XPRBctr.getRHS, 175
- XPRBctr.getRNG, 175

XPRBctr.getRowNum, 176
XPRBctr.getSlack, 176
XPRBctr.getType, 176
XPRBctr.isModCut, 177
XPRBctr.isValid, 177
XPRBctr.print, 177
XPRBctr.reset, 177
XPRBctr.setModCut, 177
XPRBctr.setRange, 178
XPRBctr.setTerm, 179
XPRBctr.setType, 179
XPRBcut, 25, 180, 181, 251
XPRBcut.add, 181
XPRBcut.addTerm, 181
XPRBcut.delTerm, 182
XPRBcut.getCRef, 182
XPRBcut.getID, 182
XPRBcut.getRHS, 182
XPRBcut.getType, 183
XPRBcut.isValid, 183
XPRBcut.print, 183
XPRBcut.reset, 183
XPRBcut.setID, 184
XPRBcut.setTerm, 184
XPRBcut.setType, 184
XPRBdefcbdelvar, 37
XPRBdefcberr, 38
XPRBdefcbmsg, 39
XPRBdelarrvar, 40
XPRBdelbasis, 41
XPRBdelctr, 42
XPRBdelcut, 43
XPRBdelcutterm, 44
XPRBdelprob, 45
XPRBdelqterm, 46
XPRBdelsos, 47
XPRBdelsosel, 48
XPRBdelterm, 49
XPRBendarrvar, 50
XPRBerror, 236
XPRBexportprob, 20, 51
XPRBexpr, 186, 187
XPRBexpr.add, 187
XPRBexpr.assign, 187
XPRBexpr.delTerm, 188
XPRBexpr.getSol, 188
XPRBexpr.mul, 188
XPRBexpr.neg, 189
XPRBexpr.setTerm, 189
XPRBfinish, 52
XPRBfixvar, 53
XPRBfree, 52
XPRBgetact, 54
XPRBgetarrvarname, 55
XPRBgetarrvarsize, 56
XPRBgetbounds, 57
XPRBgetbyname, 58
XPRBgetcolnum, 59, 246
XPRBgetctrname, 60
XPRBgetctrrng, 61
XPRBgetctrtype, 62
XPRBgetcutid, 63
XPRBgetcutrhs, 64
XPRBgetcuttype, 65
XPRBgetdual, 66
XPRBgetidxel, 67
XPRBgetidxelname, 68
XPRBgetidxsetname, 69
XPRBgetidxsetsize, 70
XPRBgetiis, 71
XPRBgetlim, 72
XPRBgetlpstat, 73
XPRBgetmipstat, 74
XPRBgetmodcut, 75
XPRBgetnumiis, 76
XPRBgetobjval, 77
XPRBgetprobname, 78
XPRBgetprobstat, 79
XPRBgetrange, 80
XPRBgetrcost, 81
XPRBgetrhs, 82
XPRBgetrownum, 83, 246
XPRBgetsense, 84
XPRBgetslack, 85
XPRBgetsol, 86
XPRBgetsosname, 87
XPRBgetsostype, 88
XPRBgettime, 89
XPRBgetvarlink, 90
XPRBgetvarname, 91
XPRBgetvarrng, 92
XPRBgetvartype, 93
XPRBgetversion, 94
XPRBgetXPRSprow, 95
XPRBidxset, 25
XPRBindexSet, 191
XPRBindexSet.addElement, 191
XPRBindexSet.getCRef, 192
XPRBindexSet.getIndex, 192
XPRBindexSet.getIndexName, 192
XPRBindexSet.getName, 193
XPRBindexSet.getSize, 193
XPRBindexSet.isValid, 193
XPRBindexSet.print, 194
XPRBindexSet.reset, 194
XPRBinit, 96, 245
XPRBlicense, 236
XPRBlicenseError, 236
XPRBloadbasis, 97
XPRBloadmat, 98, 246
XPRBmaxim, 99
XPRBminim, 100
XPRBnewarrsum, 101
XPRBnewarrvar, 102
XPRBnewctr, 103
XPRBnewcut, 104
XPRBnewcutarrsum, 105
XPRBnewcutprec, 106
XPRBnewcutsum, 107
XPRBnewidxset, 108
XPRBnewname, 109
XPRBnewprec, 110
XPRBnewprob, 96, 111, 245
XPRBnewsos, 112
XPRBnewsosrc, 113
XPRBnewsosw, 114
XPRBnewsum, 115
XPRBnewvar, 116

- XPRBprintarrvar, 117
- XPRBprintctr, 20, 118
- XPRBprintcut, 119
- XPRBprintf, 120
- XPRBprintidxset, 121
- XPRBprintobj, 20, 122
- XPRBprintprob, 20, 123
- XPRBprintsos, 124
- XPRBprintvar, 125
- XPRBprob, 25, 195, 197
- XPRBprob.addCuts, 197
- XPRBprob.clearDir, 198
- XPRBprob.delCtr, 198
- XPRBprob.delCut, 198
- XPRBprob.delSos, 199
- XPRBprob.exportProb, 199
- XPRBprob.getCRef, 199
- XPRBprob.getCtrByName, 200
- XPRBprob.getIndexSetByName, 200
- XPRBprob.getLPStat, 200
- XPRBprob.getMIPStat, 201
- XPRBprob.getName, 201
- XPRBprob.getNumIIS, 201
- XPRBprob.getObjVal, 202
- XPRBprob.getProbStat, 202
- XPRBprob.getSense, 202
- XPRBprob.getSosByName, 203
- XPRBprob.getVarByName, 203
- XPRBprob.getXPRSProb, 203
- XPRBprob.loadBasis, 204
- XPRBprob.loadMat, 204
- XPRBprob.maxim, 205
- XPRBprob.minim, 205
- XPRBprob.newCtr, 206
- XPRBprob.newCut, 206
- XPRBprob.newIndexSet, 207
- XPRBprob.newSos, 208
- XPRBprob.newVar, 208
- XPRBprob.print, 209
- XPRBprob.printObj, 209
- XPRBprob.reset, 210
- XPRBprob.saveBasis, 210
- XPRBprob.setColOrder, 210
- XPRBprob.setCutMode, 211
- XPRBprob.setDictionarySize, 211
- XPRBprob.setMsgLevel, 212
- XPRBprob.setObj, 212
- XPRBprob.setRealFmt, 213
- XPRBprob.setSense, 213
- XPRBprob.solve, 213
- XPRBprob.sync, 214
- XPRBprob.writeDir, 215
- XPRBreadarrlinecb, 126
- XPRBreadlinecb, 127
- XPRBrelation, 216
- XPRBrelation.getType, 216
- XPRBresetprob, 128
- XPRBsavebasis, 129
- XPRBsetarrvarel, 130
- XPRBsetcolorder, 131
- XPRBsetctrtype, 132
- XPRBsetcutid, 133
- XPRBsetcutmode, 134
- XPRBsetcutterm, 135

- XPRBsetcuttype, 136
- XPRBsetdecsign, 137
- XPRBsetdictionarysize, 138
- XPRBseterrctrl, 139
- XPRBsetlb, 140
- XPRBsetlim, 141
- XPRBsetmodcut, 142
- XPRBsetmsglevel, 143
- XPRBsetobj, 20, 144
- XPRBsetqterm, 20, 145
- XPRBsetrange, 146
- XPRBsetrealfmt, 147
- XPRBsetsense, 148
- XPRBsetsosdir, 149
- XPRBsetterm, 150
- XPRBsetub, 151
- XPRBsetvardir, 152
- XPRBsetvarlink, 153
- XPRBsetvartype, 154
- XPRBsolve, 155
- XPRBsos, 25, 217
- XPRBsos.add, 218
- XPRBsos.addElement, 218
- XPRBsos.delElement, 218
- XPRBsos.getCRef, 219
- XPRBsos.getName, 219
- XPRBsos.getType, 219
- XPRBsos.isValid, 219
- XPRBsos.print, 220
- XPRBsos.setDir, 220
- XPRBstartarrvar, 156
- XPRBsync, 157
- XPRBvar, 25, 221, 222
- XPRBvar.fix, 222
- XPRBvar.getColNum, 222
- XPRBvar.getCRef, 222
- XPRBvar.getLB, 223
- XPRBvar.getLim, 223
- XPRBvar.getName, 223
- XPRBvar.getRCost, 224
- XPRBvar.getRNG, 224
- XPRBvar.getSol, 225
- XPRBvar.getType, 225
- XPRBvar.getUB, 226
- XPRBvar.isValid, 226
- XPRBvar.print, 226
- XPRBvar.setDir, 226
- XPRBvar.setLB, 227
- XPRBvar.setLim, 227
- XPRBvar.setType, 228
- XPRBvar.setUB, 228
- XPRBwritedir, 158
- XPRSalter, 245
- XPRsbtran, 245
- XPRsftran, 245
- XPRSgetintattrib, 244
- XPRSgetintcontrol, 244
- XPRSgetlpsol, 244, 247
- XPRSgetmipsol, 244, 247
- XPRSglobal, 245
- XPRSiis, 244
- XPRSinit, 245
- XPRSloadbasis, 245
- XPRSloadadds, 245

XPRsloadglobal, 245
XPRsloadlp, 245
XPRsloadqp, 245
XPRsloadsecurevecs, 245
XPRsmaxim, 245
XPRsminim, 245
XPRsrange, 244
XPRsreadbasis, 245
XPRsreaddir, 245
XPRsreadprob, 245
XPRsrestore, 245
XPRssave, 244
XPRsscale, 245
XPRssetcbmessage, 39, 244
XPRssetintcontrol, 244
XPRssetprobname, 245
XPRswritebasis, 244
XPRswriteomni, 244
XPRswriteprob, 244
XPRswriteprtrange, 244
XPRswriteprtsol, 244
XPRswriterange, 244
XPRswritesol, 244