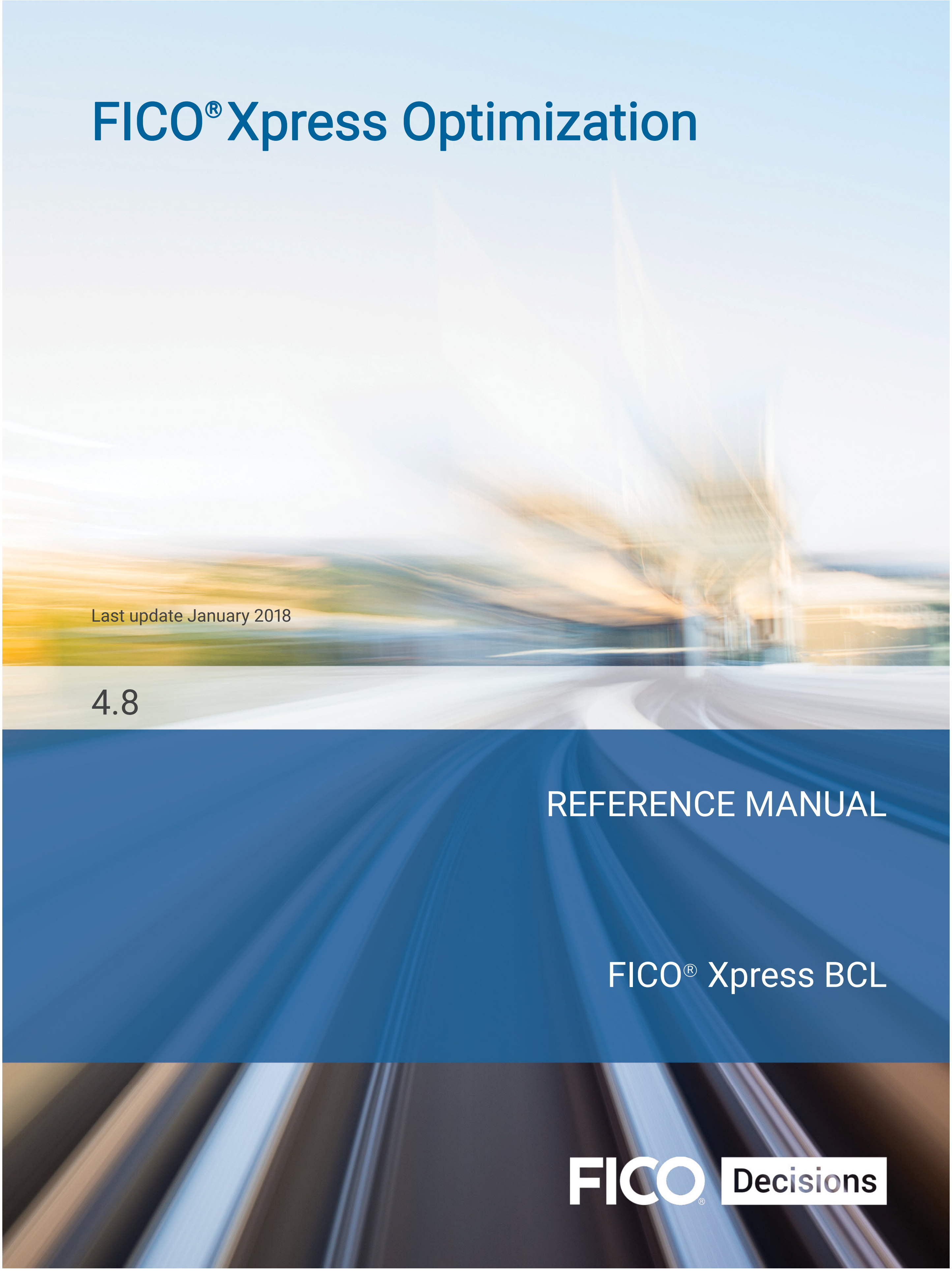

FICO® Xpress Optimization

Last update January 2018

4.8

REFERENCE MANUAL

FICO® Xpress BCL

FICO Decisions

FICO® Xpress BCL

Deliverable Version: A

Last Revised: January 2018

Version 4.8

Contents

CHAPTER 1

Introduction

1.1 An overview of Xpress BCL

The FICO® Xpress BCL Builder Component Library provides an environment in which the Xpress 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 FICO® 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 `XPRBlpoptimize` and `XPRBmipoptimize` 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, quadratic programming and user error handling 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 Chapters 5, 6, and 7 by a documentation of the C++ interface and summary descriptions of the Java and .NET interfaces. A list of BCL error and return codes and an overview of usage of BCL with the Xpress Optimizer library form the Appendices to the manual.

Please note that the full documentation of the Java and .NET interfaces is provided seperately, see subdirectories `docs/bcl/dhtml/javadoc` and `docs/bcl/bcl.net/HTML` of the Xpress installation directory.

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

This chapter gives an introduction to common BCL functions using BCL in the C language. Chapters 5 and 6 provide versions of the program examples in this chapter for the C++ and Java interfaces of BCL respectively. Available functionality for building up constraint expressions with the object-oriented interfaces differs from what is shown here for C and there are also some specifities concerning the initialization of BCL with the different interfaces.

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 (for further detail on using Optimizer functionality with BCL please refer to Appendix B). The initialization function checks for any necessary libraries, and runs security checks to determine license information about your Xpress 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

Initialize a new model	`XPRBprob pb1;`
	`...`
	`pb1 = XPRBnewprob("Problem1");`

Delete problem definition	`XPRBdelprob(pb1);`
Delete solution information	`XPRBresetprob(pb1);`
Load problem matrix	`XPRBloadmat(pb1);`
Fix column ordering	`XPRBsetcolorder(pb1,1);`
Get problem name	`XPRBgetprobname(pb1);`
Change problem name	`XPRBsetprobname(pb1,"ProbOne");`

Figure 2.1: Creating, accessing and deleting problems in BCL

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).

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, and with `XPRBsetprobname` it can be changed.

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 3.6 in the next chapter shows an example of a message callback.

The formating 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 `XPRBreadlinecb` and `XPRBreadarrlinecb`), 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`).

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

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

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 3.6 in the next chapter shows an example of an error callback.

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 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

Creating variables	`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);`

Creating variable arrays	`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);`

Accessing variables	`double ubd, lbd, lim;`
	`XPRBgetvarname(y);`
	`XPRBgetvartype(s[1]);`
	`XPRBgetcolnum(av2[0]);`
	`XPRBgetbounds(y,&lbd,&ubd);`
	`XPRBgetlim(y,&lim);`
	`XPRBsetvartype(av1[1],XPRB_BV);`

Accessing arrays	`XPRBgetarrvarname(av2);` `XPRBgetarrvarsize(av1);`
Delete a variable array	`XPRBdelarrvar(av2);`

Find by name	`XPRBvar y1; XPRBarrvar a1;`
	`y1 = XPRBgetbyname(prob,"y",XPRB_VAR);`
	`a1 = XPRBgetbyname(prob,"a1",XPRB_ARR);`

Branching directives	`XPRBsetvardir(s[0],PR,1);` `XPRBcleardir(prob);`

Setting bounds	`XPRBsetlb(y,4);`
	`XPRBsetub(s[0],9);`
	`XPRBfixvar(av[2],6);`
	`XPRBsetlim(y,5);`

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

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 at the first available free position by `XPRBapparrvarel`. For instance, assume we have defined four continuous variables `s[0]`,..., `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

$\sum_{i=0}^{3} s_i \leq 20$	`XPRBctr ctr`

`XPRBnewsum(prob,"S1",s,XPRB_L,20);`	`ctr = XPRBnewctr(prob,"S1",XPRB_L);`
	`for(i=0;i<=3;i++)`
	`XPRBaddterm(ctr,s[i],1);`
	`XPRBaddterm(ctr,NULL,20);`

$\sum_{i=0}^{3} D_i \cdot s_i = 9$	

`XPRBnewarrsum(prob,"S2",s,D, XPRB_E,9);`	`ctr = XPRBnewctr(prob,"S2",XPRB_E);`
	`XPRBaddarrterm(ctr,s,D);`
	`XPRBaddterm(ctr,NULL,9);`

$s_0 + D_0 \leq y$	$(s_0 - y \leq -D_0)$

`XPRBnewprec(prob,"Prc",s[0],D[0],y);`	`ctr=XPRBnewctr(prob,"Prc",XPRB_L);`
	`XPRBaddterm(ctr,s[0],1);`
	`XPRBaddterm(ctr,y,-1);`
	`XPRBaddterm(ctr,NULL,-D[0]);`

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

name of an array (XPRBgetarrvarname) and its size, that is, the number of variables it contains (XPRBgetarrvarsize).

Note: all variables that are added to an array of variables must belong to the same problem as the array itself.

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.

Currently defined coefficients of linear constraint terms can be retrieved with XPRBgetcoeff, for a single term, or the terms can be enumerated with XPRBgetnextterm.

Since all functions for constraint definition identify the corresponding constraint via its model 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 and function XPRBgetnextctr can be used to enumerate all constraints defined in a problem.

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 size (XPRBgetctrsize), 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. Similarly, a constraint can be transformed into a delayed row, include vars or indicator constraint with functions XPRBsetdelayed, XPRBsetincvars or XPRBsetindicator; and it can be checked if a constraint is of these types with functions XPRBgetdelayed, XPRBgetincvars and XPRBgetindicator.

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 3.5 for further detail.

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

Set objective function	`XPRBctr c;` `XPRBsetobj(prob,c);`
Set objective sense	`XPRBsetsense(prob,XPRB_MAXIM);`
Access objective sense	`int dir;` `dir = XPRBgetsense(prob);`
Locate constraint	`XPRBctr c;` `c = XPRBgetbyname(prob,"Sum1",XPRB_CTR);`
Enumerate constraints	`XPRBctr c = NULL;` `c = XPRBgetnextctr(prob,c);`
Define range constraint	`XPRBsetrange(c,1,5,15);`
Delete a constraint	`XPRBdelctr(c);`
Delete a constraint term	`XPRBvar y;` `XPRBdelterm(c,y);`

Accessing constraints	`double bdl, bdu;`
	`XPRBgetctrname(c);`
	`XPRBgetrange(c,&bdl,&bdu);`
	`XPRBgetrownum(c);`
	`XPRBgetctrsize(c);`
	`XPRBgetctrtype(c);`
	`XPRBsetctrtype(c,XPRB_L);`

Enumerate constraint terms	`double coeff; const void *ref;` `ref = XPRBgetnextterm(c,ref,&y,&coeff);`

Special constraints	`XPRBgetmodcut(c);`
	`XPRBsetmodcut(c,1);`
	`XPRBgetdelayed(c);`
	`XPRBsetdelayed(c,1);`
	`XPRBgetincvars(c);`
	`XPRBsetincvars(c,1);`
	`XPRBgetindicator(c);`
	`XPRBgetindvar(c);`
	`XPRBsetindicator(c,1,y);`

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

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).

Solve active problem	`XPRBlpoptimize(prob,"p");` `XPRBmipoptimize(prob,"");`

Status information	`XPRBgetprobstat(prob);`
	`XPRBgetlpstat(prob);`
	`XPRBgetmipstat(prob);`

Get objective value	`XPRBgetobjval(prob);`

Load solutions	`int ncol; double *d; XPRBsol s;`
	`XPRBloadmipsol(prob,d,ncol,1);`
	`XPRBaddmipsol(prob,s);`

Solution reading and writing	`XPRBreadbinsol(prob,"","");`
	`XPRBreadslxsol(prob,"","");`
	`XPRBwritesol(prob,"","");`
	`XPRBwritebinsol(prob,"","");`
	`XPRBwriteprtsol(prob,"","");`
	`XPRBwriteslxsol(prob,"","");`

Solution information	`XPRBvar y; XPRBctr c;`
	`XPRBgetsol(y); XPRBgetdual(c);`
	`XPRBgetrcost(y); XPRBgetslack(c);`
	`XPRBgetact(c);`

Ranging information	`XPRBgetvarrng(y,XPRB_UCOST);` `XPRBgetctrrng(c,XPRB_LOACT);`
Fix globals	`XPRBfixglobals(prob,1);`

Advanced bases	`XPRBbasis b;`
	`b=XPRBsavebasis(prob);`
	`XPRBloadbasis(b);`
	`XPRBdelbasis(b):`

Figure 2.6: Solving and solution information

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.

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 (`XPRBlpoptimize`, `XPRBmipoptimize`) 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`. It is also required to set the sense of the objective, that is, whether to minimize (default) or to maximize the objective. All solution functions `XPRBlpoptimize`, and `XPRBmipoptimize` 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, LP only), constraint activity (XPRBgetact), and dual values (XPRBgetdual, LP only). It is also possible to obtain ranging information for variables (XPRBgetvarrng) and constraints (XPRBgetctrrng) 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 or XPRB_XPRS_SOLMIP (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. It is possible to force the loading of some variables into the Optimizer by creating a type XPRB_N constraint, adding the variables to this constraint (with any non-zero coefficients) and setting the constraint to be an *include vars* constraint with XPRBsetincvars. *Include vars* constraints are not loaded into the Optimizer but all variables they contain are always loaded (even if they don't appear in other constraints).

When solving MIP problems, the function XPRBfixglobals fixes all the global entities to the values of the last found MIP solution. This is useful for example for finding the reduced costs of the continuous variables after the global variables have been fixed to their optimal values.

It is possible to load solutions from memory with function XPRBloadmipsol, which loads a solution into BCL or the Optimizer, or XPRBaddmipsol which can also load partial or infeasible solutions into the Optimizer. Solutions can also be written and read from file with XPRBwritesol, XPRBwritebinsol, XPRBwriteprtsol, XPRBwriteslxsol, XPRBreadslxsol.

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 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 startj 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);
 XPRBmipoptimize(prob, "");       /* 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 `XPRBreadarrlinecb` and `XPRBreadlinecb` 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, `XPRBreadlinecb` 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 Language 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

Data input from file

```
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);
```

Create a new index set

```
XPRBidx set1;
set1 = XPRBnewidxset(prob,"Set1",100);
```

Add index to a set

```
XPRBaddidxel(set1,"Prob1");
```

Accessing index sets

```
int size, ind;
ind = XPRBgetidxel(set1,"Prod1");
name = XPRBgetidxelname(set1,14);
name = XPRBgetidxsetname(set1);
size = XPRBgetidxsetsize(set1);
```

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

the jobs, such as ABC14, DE45F, GH9IJ99, KLMN789. We further assume that these 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 `XPRBreadlinecb`, 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;
}
```

	`XPRBsos set1, set2;` `XPRBarrvar s;`
Immediate (ref. constraint)	`XPRBctr c;` `set1=XPRBnewsosrc(prob,"sA",XPRB_S2,s,c);`
Immediate (coefficients)	`double C[] = 1,2,3,4;` `set2=XPRBnewsosw(prob,"sB",XPRB_S1,s,C);`
Consecutive definition	`set2=XPRBnewsos(prob,"sB",XPRB_S1);` `XPRBaddsosarrel(set2,s,C);`
Delete set definition	`XPRBdelsos(set2);`

Accessing sets	`XPRBaddsosel(set2,s[2],4,5);`
	`XPRBdelsosel(set1,s[0]);`
	`XPRBgetsosname(set1);`
	`XPRBgetsostype(set2);`

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

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 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.

Note: all members that are added to a SOS must belong to the same problem as the SOS itself.

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 Loading solutions

3.3.1 Basic functions

With the Xpress Optimizer, it is possible to load external MIP solutions from memory to help the global search and heuristic algorithms. BCL replicates this same functionality with the `XPRBloadmipsol` and `XPRBaddmipsol` functions. In both cases, with *solution* we mean just the variable solution values: no objective value, slack or dual information is passed to the Optimizer as part of the solutions . The first function, `XPRBloadmipsol`, can be used when the solution values are known for all the model variables and the solution is feasible. The second one, XPRBaddmipsol, can be used even when only a partial solution is known or the "solution" is not feasible. If the provided solution is a partial solution or found to be infeasible, the Optimizer will run a limited local search heuristic in an attempt to find a close feasible integer solution.

In order to be loaded with XPRBaddmipsol, a solution must first be defined as a XPRBsol object. A new XPRBsol solution can be created with the XPRBnewsol function. Individual variables can then be added with function XPRBsetsolvar and arrays of variables with function XPRBsetsolarrvar, each time indicating the corresponding solution values. The current solution size can be obtained with XPRBgetsolsize and assigned variables can be queried or removed with functions XPRBgetsolvar and XPRBdelsolvar. Finally, the entire solution object can be deleted with function XPRBdelsol.

Note that it is also possible load external solutions from file with the XPRBreadbinsol and XPRBreadslxsol functions.

Note: all variables that are added to a solution must belong to the same problem as the solution itself.

3.3.2 Example

The following code fragment shows how to create a partial solution and load it into the Optimizer with XPRBaddmipsol. It also shows how to set a callback function to receive the notification from the Optimizer on the solution loading outcome, which informs the user if the solution has been accepted and if it has been found to be feasible or required a reoptimization or local search heuristic (note that defining this callback is optional). The complete example can be found in file d1wagon2.c.

```
/* Callback function reporting loaded solution status */
void XPRS_CC solnotify(XPRSprob my_prob, void* my_object, const char* solname, int status)
{
  printf("Optimizer loaded solution %s with status=%d\n", solname, status);
}

void d1w2_solve(XPRBprob prob)
{
  int b;
  XPRBsol sol;
  ...

  /* Create a BCL solution from the heuristic solution previously found */
  sol = XPRBnewsol(prob);

  /* Set the solution values for some discrete variables */
  for (b = 0; b < NBOXES; b++) XPRBsetsolvar(sol, load[b][HeurSol[b]], 1);

  /* Send the solution to the optimizer */
  XPRBaddmipsol(prob, sol, "heurSol");

  /* Free the solution object */
  XPRBdelsol(sol);

  /* Request notification of solution status after processing */
  XPRSaddcbusersolnotify(XPRBgetXPRSprob(prob), solnotify, NULL, 0);
  ...
}
```

3.4 Output and printing

BCL provides printing functions for variables, constraints, Special Ordered Sets, and index sets (XPRBprintvar, XPRBprintarrvar, XPRBprintctr, XPRBprintsos, XPRBprintidxset, XPRBprintsol) as well as the entire model definition (XPRBprintprob). Any program output may be

File output	`XPRBexportprob(prob,XPRB_MPS,"expl2");`

Print model objects	`XPRBvar y;`
	`XPRBprintvar(y);`
	`XPRBarrvar av;`
	`XPRBprintarrvar(av);`
	`XPRBctr c;`
	`XPRBprintctr(c);`
	`XPRBsos s;`
	`XPRBprintsos(s);`
	`XPRBidxset is;`
	`XPRBprintidxset(is);`
	`XPRBsol sol;`
	`XPRBprintsol(sol);`

Print a given problem	`XPRBprintprob(prob);`
Print program output	`XPRBprintf("Print this text");`
Compose a name string	`int i = 3;` `XPRBnewname("abc%d",i);`

Figure 3.3: File output and printing.

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.

3.4.1 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 */
```

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)
{
```

Add quadratic term	`XPRBctr c;`
	`XPRBvar x1;`
	`XPRBaddqterm(c,x1,x1,3);`

Set quadratic term	`XPRBvar x2;` `XPRBsetqterm(c,x1,x2,-7.2);`
Delete a quadratic term	`XPRBdelqterm(c,x2,x1);`

Enumerate quadratic terms	`double coeff;`
	`const void *ref;`
	`ref = XPRBgetnextqterm(c,ref,&x1,&x1,&coeff);`

Figure 3.4: Defining and accessing quadratic terms in BCL.

```
  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);
  ...
 }
```

3.5 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 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 objevctive 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.

Any other quadratic form supported by the Xpress Optimizer can also be used, e.g. Second Order Cone constraints in Second Order Cone problems (SOCPs).

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.

The coefficient of a quadratic constraint term can be retrieved with XPRBgetqcoeff. The quadratic terms of a constraint can be enumerated with XPRBgetnextqterm.

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.

3.5.1 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.

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

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 */
 XPRBlpoptimize(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;
}
```

3.6 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 */

 XPRBlpoptimize(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.

3.7 Efficent modeling with BCL

This section discusses some recommendations for the efficient use of BCL. Such considerations are particularly important when working with large-size optimization problems or when solving a large number of models / model instances in a single application. Our criteria for measuring efficiency are:

- model execution speed
- memory consumption

Please note that this section is only concerned with modeling aspects. For issues relating to the solving process, such as the performance of the underlying optimization algorithms, the reader is refered to the *Xpress Optimizer Reference Manual*.

3.7.1 Names dictionaries

BCL works with two names dictionaries, the main names dictionary (storing the names of constraints, decision variables, *etc.*) and a dedicated dictionary for index set elements. The former is active by default wheras the latter gets activated only if a model uses index sets. The following remarks refer principally to the names dictionary.

3.7.1.1 Disabling the names dictionary

If an application does not make use of the names of modeling objects the names dictionary can be disabled to save memory. The function `XPRBsetdictionarysize` for resetting the dictionary size can only be called immediately after the creation of the corresponding problem. Once the dictionary 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.

- *C:* `XPRBsetdictionarysize(prob, XPRB_DICT_NAMES, 0);`
- *C++:* `XPRBprob.setDictionarySize(XPRB_DICT_NAMES, 0);`
- *Java:* `XPRBprob.setDictionarySize(XPRB.DICT_NAMES, 0);`
- *C#:* `XPRBprob.setDictionarySize(BCLconstant.DICT_NAMES, 0);`

3.7.1.2 Setting the names dictionary size

If you wish to use the names dictionary we recommend to choose a size close to the number of variables+constraints in your problem, preferrably a prime number. (Too small values will slow down access to the names dictionary, larger values imply higher memory usage.)

3.7.2 Handling of problems

3.7.2.1 Resetting a problem

You should reset a problem to free up memory if the solution information is no longer required (function `XPRBresetprob`). Resetting a problem 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.

- *C:* `XPRBresetprob(prob);`
- *C++/Java/C#:* `XPRBprob.reset();`

Other functions for freeing memory of auxiliary/intermediate structures:
`XPRBcleardir`, `XPRBdelarrvar`, `XPRBdelbasis`, `XPRBdelcut`

3.7.2.2 Releasing a problem

With C a problem may be deleted explicictly (`XPRBdelprob`) to free up all memory used by it. In the object-oriented interfaces make sure to release all references to a problem to enable garbage collection on the object.

The Java interface also publishes the problem finalizer: `XPRBprob.finalize()`.

3.7.3 Constraint definition

3.7.3.1 Object-oriented interfaces

Overloaded operators and the more algebraic-style definition of constraints via expressions in the object-oriented interfaces of BCL lead to more easily human-readable models but unfortunately, they also create many intermediate objects, making them computationally less efficient. With constraint/expression sizes upwards of 1000 terms a slowdown tends to become noticeable and alternative ways of constraint formulation should be sought.

The best alternative is to use the `addTerm` and `setTerm` methods for constraints or expressions (these avoid the creation of intermediate objects, such as terms or expressions, thus reducing memory consumption and most often leading to a speed up).

Example:

- *C++:*
  Replace
  ```
  ctr += 17*x;
  ```
  by
  ```
  ctr.addTerm(17, x);
  ```
- *Java:*
  Replace
  ```
  ctr.add(x.mul(17));
  ```
  by
  ```
  ctr.addTerm(x, 17); // or: ctr.addTerm(17, x);
  ```

3.7.3.2 Order of enumeration

In pre-Release 2008 versions of BCL it is recomended to enumerate / access decision variables within loops in the order of their creation. This recommendation does *not* apply to BCL 4.0 and newer.

II. BCL library and class reference

CHAPTER 4

BCL C library functions

A large number of routines are available within the Xpress 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;
`XPRBsol`	a solution;
`XPRBsos`	a Special Ordered Set (SOS1 of 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 `NULL` 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

`ctr` Reference to a constraint.

`av` Reference to an array of variables.

`coeff` 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 `ctr1`.

```
XPRBprob prob;
XPRBctr ctr1;
XPRBarrvar ty1;
double cr[] = {2, 13, 15, 6, 8.5};
 ...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBaddarrterm(ctr1, 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.

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

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

`cut`	Reference to a cut.
`av`	Reference to an array of variables.
`coeff`	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 *cut*1.

```
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.

Note: all terms that are added to a cut must belong to the same problem as the cut itself.

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. The function call is surrounded by the pair `XPRBbegincb` and `XPRBendcb` to coordinate BCL with the local Optimizer subproblem in the case of a multi-threaded 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)
  {
    XPRBbegincb(expl1, oprob);
    XPRBaddcuts(expl1, ca, 3);
    XPRBendcb(expl1);
  }
  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 */
  XPRBmipoptimize(expl1, "");           /* 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

`XPRBbegincb`, `XPRBendcb`, `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 $5.4 \cdot x1$ to the cut *cut*1.

```
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.

Note: all terms that are added to a cut must belong to the same problem as the cut itself.

Related topics

`XPRBnewcut`, `XPRBaddcutarrterm` `XPRBdelcutterm`, `XPRBsetcutterm`.

XPRBaddidxel

Purpose

Add an index to an index set.

Synopsis

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

Arguments

`idx`	A BCL index set.
`name`	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`.

XPRBaddmipsol

Purpose

Add a new feasible, infeasible or partial MIP solution to the Optimizer.

Synopsis

```
int XPRBaddmipsol(XPRBprob prob, XPRBsol sol, const char *name);
```

Arguments

`prob`	Reference to a problem.
`sol`	Reference to a solution.
`name`	An optional name to associate with the solution. Can be NULL.

Return value

0 if function executed successfully, 1 otherwise.

Example

Add a MIP solution for problem `expl2` to the Optimizer.

```
XPRBprob expl2;
XPRBsol sol;
   ...
XPRBaddmipsol(prob, sol, "myHeurSol");
XPRBdelsol(sol); /* Free the solution object */
/* Request notification of solution status after processing */
XPRSaddcbusersolnotify(XPRBgetXPRSprob(prob), solnotify, NULL, 0);
```

Further information

1. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition, it is regenerated automatically before adding the solution.
2. The `XPRBmipoptimize` function by default resets problem status including eventual loaded solutions; to avoid that, flag "c" should be specified for the `alg` argument of `XPRBmipoptimize` when called after `XPRBaddmipsol`.
3. The function returns immediately after passing the solution to the Optimizer. The solution is placed in a pool until the Optimizer is able to analyze the solution during a MIP solve.
4. If the provided solution is partial or found to be infeasible, a limited local search heuristic will be run in an attempt to find a close feasible integer solution. Values provided for continuous columns in partial solutions are currently ignored.
5. The *usersolnotify* callback can be used to discover the outcome of a loaded solution. The optional name provided as `name` will be returned in the callback.

Related topics

`XPRBloadmipsol`, `XPRBloadmat`.

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 `var1 * var2`.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following example adds the term `-2*x2*x4` to the constraint `ctr1`:

```
XPRBctr ctr1;
XPRBvar x2,x4;
 ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_L);
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
x4 = XPRBnewvar(prob, XPRB_PL, "abc5",0 , XPRB_INFINITY);
XPRBaddqterm(ctr1, 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.

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

Related topics

`XPRBdelqterm`, `XPRBsetqterm`.

XPRBaddsosarrel

Purpose

Add multiple elements to a SOS.

Synopsis

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

Arguments

`sos`	A SOS of type 1 or 2.
`av`	An array of variables.
`weight`	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`.

Note: all members that are added to a SOS must belong to the same problem as the SOS itself.

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.

Note: all members that are added to a SOS must belong to the same problem as the SOS itself.

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 ctr1;
XPRBvar x1;
   ...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBaddterm(ctr1, x1, 5.4);
```

This adds the term `5.4*x1` to the constraint `ctr1`.

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.

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

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
`av`	BCL reference to an array.
`var`	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.

Note: all variables that are added to an array of variables must belong to the same problem as the array itself.

Related topics
`XPRBdelarrvar`, `XPRBendarrvar`, `XPRBnewarrvar`, `XPRBsetarrvarel`, `XPRBstartarrvar`.

XPRBbegincb

Purpose
Start using the local optimizer problem with BCL in a callback.

Synopsis
```
int XPRBbegincb(XPRBprob bprob, XPRSprob oprob);
```

Arguments
`bprob` Reference to a BCL problem.

`oprob` Reference to an Xpress Optimizer problem.

Return value
0 if function executed successfully, 1 otherwise.

Example
The example shows how to set up the integer solution callback of Xpress Optimizer to use BCL to display the results.

```
XPRBprob bprob;
XPRBvar x;

void XPRS_CC printsol(XPRSprob oprob, void *my_object){
  int num;

  XPRSgetintattrib(oprob, XPRS_MIPSOLS, &num);
                                 /* Get number of the solution */

  XPRBbegincb(bprob, oprob);       /* Use local Optimizer problem */
  XPRBsync(bprob, XPRB_XPRS_SOL);  /* Update BCL solution values */
  XPRBprintf(bprob, "Solution %d: Objective value: %g\n",
             num, XPRBgetobjval(bprob));
  XPRBprintf(bprob, "%s: %g\n", XPRBgetvarname(x), XPRBgetsol(x));
  XPRBendcb(bprob);                /* Reset BCL to main problem */
}

int main(int argc, char **argv)
{
  XPRSprob oprob;
  bprob = XPRBnewprob("cbexample");
  ...                                      /* Define the problem */
  oprob = XPRBgetXPRSprob(bprob);          /* Get Optimizer problem */
  XPRSsetcbintsol(oprob, printsol, NULL);  /* Set the callback */
  XPRBmipoptimize(bprob, "");              /* Solve the MIP problem */
}
```

Further information
This function switches from the original problem to the specified (local) optimizer problem for all BCL accesses to Xpress Optimizer (in particular, for solution updates and the definition of cuts). A call to this function must precede any call to such BCL functions in optimizer MIP callbacks when the default multi-threaded MIP search is used for solving a problem. A call to `XPRBbegincb` must always be matched by a call to `XPRBendcb` to reset the optimizer problem within BCL and to release the BCL problem (access to the BCL problem is locked to the particular thread in between the two function calls).

Related topics
`XPRBendcb`, `XPRBgetsol`, `XPRBaddcuts`.

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. If this function is called after the creation of a problem, then the version without problem pointer will only influence the global BCL error handling settings, to change the settings for the specific problem you need to use the problem as its first argument. Global error handling settings apply to all cases of errors where no problem information is known, for example in the case of a missing/uninitialized model object used as argument to certain BCL functions.

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. If this function is called after the creation of a problem, then the version without problem pointer will only influence the global BCL message handling settings, to change the settings for the specific problem you need to use the problem as its first argument. Global message handling settings apply to all situations where no problem information is available at the point from where the output is produced, otherwise the problem-specific message handling will be used.

3. A BCL program must *not* reset the message callback `XPRSsetcbmessage` of Xpress Optimizer—please use the callback chaining function `XPRSaddcbmessage` (however, all other logging callbacks of the Optimizer may be defined either way).

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");
   ...
XPRBlpoptimize(expl2, "");
basis = XPRBsavebasis(expl2);
   ...
XPRBloadmat(expl2);
XPRBloadbasis(basis);
XPRBdelbasis(basis);
XPRBlpoptimize(expl2,"");
```

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 ctr1;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBdelctr(ctr1);
```

This deletes the constraint `ctr1`.

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`. If the constraint occurs in a previously saved basis that is to be (re)loaded later on you should change its type to `XPRB_N` using `XPRBsetctrtype` instead of entirely deleting the constraint.

Related topics

`XPRBnewctr`, `XPRBsetctrtype`, `XPRBloadbasis`.

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 *cut*1.

```
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

`cut`	Reference to a cut as resulting from `XPRBnewcut`.
`var`	Reference to a variable in the cut.

Return value

0 if function executed successfully, 1 otherwise.

Example

Add the term $5.4 \cdot x1$ to the cut *cut*1 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*x2*x2` to the constraint `ctr1` and then deletes this term from the constraint:

```
XPRBctr ctr1;
XPRBvar x2,x4;
 ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_L);
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
XPRBaddqterm(ctr1, x2, x2, 5.2);
XPRBdelqterm(ctr1, 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`.

XPRBdelsol

Purpose

Delete a solution.

Synopsis

```
int XPRBdelsol(XPRBsol sol);
```

Argument

`sol` Reference to a previously defined solution.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following deletes the Ssolution `sol1`.

```
XPRBprob prob;
XPRBsol sol1;
   ...
sol1 = XPRBnewsol(prob);
XPRBdelsol(sol1);
```

Further information

This function deletes an `XPRBsol` solution (without deleting the variables it contains).

Related topics

`XPRBdelsolvar`, `XPRBnewsol`, `XPRBsetsolarrvar`, `XPRBsetsolvar`.

XPRBdelsolvar

Purpose

Delete a variable from a solution.

Synopsis

```
int XPRBdelsolvar(XPRBsol, XPRBvar var);
```

Arguments

`sol`	BCL reference to a previously created solution.
`var`	BCL reference to a variable.

Return value

0 if function executed successfully, 1 otherwise.

Example

This code deletes the variable `x1` from the solution `sol1`.

```
XPRBprob prob;
XPRBsol1 sol1;
XPRBvar x1;
   ...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);
sol1 = XPRBnewsol(prob);
XPRBsetsolvar(sol1, x1, 5.4);
   ...
XPRBdelsolvar(sol1, x1);
```

Further information

This function deletes a variable (assigned to a value) from the given solution.

Related topics

`XPRBdelsol`, `XPRBnewsol`, `XPRBsetsolarrvar`, `XPRBsetsolvar`.

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`.

XPRBdelsosel

Purpose

Delete an element from a SOS.

Synopsis

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

Arguments

`sos`	A SOS of type 1 or 2.
`var`	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);
XPRBaddsosel(set1, x2, 9.0);
XPRBdelsosel(set1, x2);
```

Further information

This function removes a variable from a Special Ordered Set.

Related topics

`XPRBaddsosarrel`, `XPRBaddsosel`, `XPRBdelsos`, `XPRBnewsos`.

XPRBdelterm

Purpose

Delete a linear term from a constraint.

Synopsis

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

Arguments

`ctr` BCL reference to a previously created constraint.

`var` 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 ctr1;
XPRBvar x1;
   ...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBaddterm(ctr1, x1, 5.4);
XPRBdelterm(ctr1, 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`.

XPRBendcb

Purpose

Reset BCL to the original optimizer problem in a callback.

Synopsis

```
int XPRBendcb(XPRBprob bprob);
```

Argument

`bprob` Reference to a BCL problem.

Return value

0 if function executed successfully, 1 otherwise.

Example

The example shows how to set up the MIP solution callback to use BCL to display the results.

```
XPRBprob bprob;
XPRBvar x;

void XPRS_CC printsol(XPRSprob oprob,void *my_object){
  int num;

  XPRSgetintattrib(oprob, XPRS_MIPSOLS, &num);
                                 /* Get number of the solution */

  XPRBbegincb(bprob, oprob);     /* Use local Optimizer problem */
  XPRBsync(bprob, XPRB_XPRS_SOL); /* Update BCL solution values */
  XPRBprintf(bprob, "Solution %d: Objective value: %g\n",
             num, XPRBgetobjval(bprob));
  XPRBprintf(bprob, "%s: %g\n", XPRBgetvarname(x), XPRBgetsol(x));
  XPRBendcb(bprob);              /* Reset BCL to main problem */
}

int main(int argc, char **argv)
{
  XPRSprob oprob;
  bprob = XPRBnewprob("cbexample");
  ...                                     /* Define the problem */
  oprob = XPRBgetXPRSprob(bprob);         /* Get Optimizer problem */
  XPRSsetcbintsol(oprob, printsol, NULL); /* Set the callback */
  XPRBmipoptimize(bprob, "");             /* Solve the MIP problem */
}
```

Further information

This function switches back to the original optimizer problem for all BCL accesses to Xpress Optimizer. A call to this function terminates a block of calls to BCL functions in an optimizer callback that is preceded by `XPRBbegincb`. The call to `XPRBendcb` releases the BCL problem (access to the BCL problem is locked to the particular thread between the two function calls).

Related topics

`XPRBbegincb`.

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.mps`.

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 `.mps`. 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.
3. The precision used by BCL for printing real numbers can be changed with `XPRBsetrealfmt` to obtain more accurate output for very large or very small numbers. For full precision matrix output the user is advised to switch to the Optimizer function `XPRSwriteprob`, preceded by a call to `XPRBloadmat` (see Appendix B for further detail).

Related topics

`XPRBprintprob`, `XPRBprintf`, `XPRBsetrealfmt`, `XPRBloadmat`.

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`.

XPRBfixglobals

Purpose
Fixes all the global entities to the values of the last found MIP solution.

Synopsis
```
int XPRBfixglobals(XPRBprob prob, int ifround);
```

Arguments

`prob`	Reference to a problem.
`ifround`	If all global entities should be rounded to the nearest discrete value in the solution before being fixed.

Return value
0 if function executed successfully, 1 otherwise.

Example
Performs a global search on problem expl2 and then uses XPRBfixglobals before solving the remaining linear problem.

```
XPRBprob expl2;
   ...
XPRBmipoptimize(expl2, "");
XPRBfixglobals(expl2, 1);
XPRBlpoptimize(expl2, "");
XPRBwriteprtsol(expl2);
```

Further information

1. This is useful e.g. for finding the reduced costs for the continuous variables after the global variables have been fixed to their optimal values. The global variables are fixed to the value of the MIP solution only in the Optimizer (not in BCL).
2. In order to eventually resync the bounds of global variables to their original values defined in BCL (i.e. unfix them), a call to `XPRBsync` with the flag `XPRB_XPRS_PROB` can be used.

Related topics
`XPRBloadmat`, `XPRBsync`.

XPRBfixvar

Purpose

Fix a variable.

Synopsis

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

Arguments

`var` BCL reference to a variable.

`val` 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);
XPRBlpoptimize(expl2, "");
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 solution is available 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 `NULL` if not required.
`bdu`	Upper bound value. May be `NULL` 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

`prob`	Reference to a problem.
`name`	The name of the object.
`type`	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`.

XPRBgetcoeff

Purpose
Get the coefficient of a linear constraint term.

Synopsis
```
double XPRBgetcoeff(XPRBctr ctr, XPRBvar var);
```

Arguments
`ctr` BCL reference to a previously created constraint.

`var` BCL reference to a variable. May be `NULL` to indicate the constant term.

Return value
Coefficient of the variable in the specified constraint or 0 if the variable does not occur.

Example
```
XPRBprob expl2;
XPRBctr ctr1;
XPRBvar x1;
double val;
   ...
expl2 = XPRBnewprob("example2");
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 0, 100);
ctr1 = XPRBnewctr(expl2, "r1", XPRB_E);
XPRBaddterm(ctr1, x1, 5.4);
val = XPRBgetcoeff(ctr1, x1);
```

This retrieves the coefficient of the variable `x1` in the constraint `ctr1`.

Further information
This function returns the coefficient of a given variable `var` in the constraint `ctr`. Return value 0 indicates that the variable is not contained in the constraint. If `var` is set to `NULL`, this function returns the right hand side (constant term) of the constraint.

Related topics
`XPRBaddterm`, `XPRBgetqcoeff`, `XPRBgetrhs`, `XPRBnewctr`, `XPRBsetterm`.

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 `XPRBgetprobstat`. 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`.

XPRBgetctrrng

Purpose

Get ranging information for a constraint.

Synopsis

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

Arguments

`ctr` Reference to a previously created constraint.

`rngtype` The type of ranging information sought. This is one of:

`XPRB_UPACT`	upper activity (= the level to which the constraint activity may be increased at a cost per unit of increase given by the XPRB_UUP value, ignoring the upper bound on the constraint as specified by its RHS);
`XPRB_LOACT`	lower activity (= the level to which the constraint activity may be decreased at a cost per unit of decrease given by the XPRB_UDN value, ignoring the lower bound on the constraint as specified by its RHS);
`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);
 ...
XPRBlpoptimize(expl2, "");
upact = XPRBgetctrrng(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`.

XPRBgetctrsize

Purpose

Get the size of a constraint.

Synopsis

```
int XPRBgetctrsize(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

Size (= number of linear or quadratic terms with a non-zero coefficient) of the constraint, or -1 in case of an error.

Example

The following returns the size of the constraint `ctr1`.

```
XPRBprob expl2;
XPRBctr ctr1;
int size;
   ...
expl2 = XPRBnewprob("example2");
ctr1 = XPRBnewctr(expl2, "r1", XPRB_E);
   ...
size = XPRBgetctrsize(ctr1);
```

Related topics

`XPRBgetctrname`, `XPRBgetctrtype`, `XPRBnewctr`.

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 `ctr1`.

```
XPRBprob expl2;
XPRBctr ctr1;
char rtype;
   ...
expl2 = XPRBnewprob("example2");
ctr1 = XPRBnewctr(expl2, "r1", XPRB_E);
rtype = XPRBgetctrtype(ctr1);
```

Further information

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

Related topics

`XPRBgetctrname`, `XPRBgetctrsize`, `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 *cut*1.

```
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 *cut*1.

```
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 *cut*1.

```
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`.

XPRBgetdelayed

Purpose

Get the *delayed* type of a constraint.

Synopsis

```
int XPRBgetdelayed(XPRBctr ctr);
```

Argument

`ctr`	Reference to a previously created constraint.

Return value

`0`	an ordinary constraint;
`1`	a delayed constraint;
`-1`	an error has occurred.

Example

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

This determines whether `ctr1` is an ordinary constraint or a delayed constraint.

Further information

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

Related topics

`XPRBsetdelayed`.

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);
XPRBlpoptimize(expl2, "");
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.
Dual information is available only after LP solving. To obtain dual values for a MIP solution (that is, if function `XPRBgetmipstat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), you need to fix the discrete variables to their solution values with a call to `XPRSfixglobals`, followed by a call to `XPRBlpoptimize` before calling `XPRBgetdual`. Otherwise, if this function is called when a MIP solution is available it returns 0.
If no solution information is available this function outputs a warning and returns 0.
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`. In this case it returns the dual value in the last LP that has been solved.

Related topics

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

XPRBgetidxel

Purpose

Get the index number of an index.

Synopsis

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

Arguments

`idx`	A BCL set name
`name`	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

`idx`	A BCL index set.
`i`	Index number.

Return value

Name of the ith 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 `NULL`).
`numv`	Reference to an integer that gets assigned the number of variables returned by the function (may be `NULL`).
`arrctr`	Reference to a table of BCL constraints (may be `NULL`).
`numc`	Reference to an integer that gets assigned the number of constraints returned by the function (may be `NULL`).
`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");
   ...
XPRBlpoptimize(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");
 XPRBfreemem(iisvar);           /* Free the array of variables */
 printf("Constraints: ");       /* Print all constraints */
 for(i=0;i<numc;i++) printf("%s ", XPRBgetctrname(iisctr[i]));
 printf("\n");
 XPRBfreemem(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 by calls to `XPRBfreemem`.

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`, `XPRBgetmiis`.

XPRBgetincvars

Purpose

Get the type of a constraint.

Synopsis

```
int XPRBgetincvars(XPRBctr ctr);
```

Argument

`ctr` Reference to a previously created constraint.

Return value

`0` an ordinary constraint;

`1` an *include vars* special constraint;

`-1` an error has occurred.

Example

```
XPRBprob prob;
XPRBctr ctr1;
int mcstat;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_N);
mcstat = XPRBgetincvars(ctr1);
```

This determines whether `ctr1` is an ordinary constraint or an *include vars* special constraint.

Further information

This function indicates whether the given constraint is an *include vars* special constraint or an ordinary constraint.

Related topics

`XPRBsetincvars`.

XPRBgetindicator

Purpose
Get the type of an indicator constraint.

Synopsis
```
int XPRBgetindicator(XPRBctr ctr);
```

Argument
`ctr`	Reference to a previously created constraint.

Return value
`0`	an ordinary constraint;
`1`	an indicator constraint with condition $b = 1$;
`-1`	an indicator constraint with condition $b = 0$;
`-2`	an error has occurred.

Example
```
XPRBprob prob;
XPRBctr ctr1;
int istat;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_L);
istat = XPRBgetindicator(ctr1);
```

This determines whether `ctr1` is an ordinary constraint or an indicator constraint.

Further information
This function indicates whether the given constraint is an indicator constraint or an ordinary constraint. In the case of an indicator constraint the return value also specifies the sense of the condition.

Related topics
`XPRBgetindvar`, `XPRBsetindicator`.

XPRBgetindvar

Purpose
Get the variable associated with an constraint.

Synopsis
```
XPRBvar XPRBgetindvar(XPRBctr ctr);
```

Argument
`ctr` Reference to a previously created constraint.

Return value
The indicator variable or `NULL` in case of an error.

Example
```
XPRBprob prob;
XPRBctr ctr1;
XPRBvar x;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_L);
if(XPRBgetindicator(ctr1)==-1)
{
  x = XPRBgetindvar(ctr1);
  printf("%s=0 -> %s\n", XPRBgetvarname(x), XPRBgetctrname(ctr1));
}
if(XPRBgetindicator(ctr1)==1)
{
  x = XPRBgetindvar(ctr1);
  printf("%s=1 -> %s\n", XPRBgetvarname(x), XPRBgetctrname(ctr1));
}
```

This prints out the name of the indicator variable associated with the indicator constraint `ctr1` and the sense of the implication.

Further information
This function returns the indicator variable associated with an indicator constraint.

Related topics
`XPRBgetindicator`, `XPRBsetindicator`.

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

`var` BCL reference to a variable.

`lim` 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`	LP problem is not solved.
`XPRB_LP_NONCONVEX`	QP problem is nonconvex;

Example

The following returns the current LP status.

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

Further information

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

Related topics

`XPRBgetmipstat`, `XPRBgetprobstat`.

XPRBgetmiis

Purpose

Get the variables, constraints, and SOS of an IIS.

Synopsis

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

Arguments

`prob`	Reference to a problem.
`arrvar`	Reference to a table of BCL variables (may be `NULL`).
`numv`	Reference to an integer that gets assigned the number of variables returned by the function (may be `NULL`).
`arrctr`	Reference to a table of BCL constraints (may be `NULL`).
`numc`	Reference to an integer that gets assigned the number of constraints returned by the function (may be `NULL`).
`arrsos`	Reference to a table of BCL SOS (may be `NULL`).
`nums`	Reference to an integer that gets assigned the number of SOS returned by the function (may be `NULL`).
`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;
XPRBvar *iissos;
int numv, numc, nums;
expl2 = XPRBnewprob("example2");
   ...
XPRBmipoptimize(expl2, "");
if(XPRBgetmipstat(expl2)==XPRB_MIP_INFEAS)
{
 XPRBgetmiis(expl2, &iisvar, &numv, &iisctr, &numc, &iissos, &nums, 1);
 printf("Variables: ");        /* Print all variables */
 for(i=0;i<numv;i++) printf("%s ", XPRBgetvarname(iisvar[i]));
 printf("\n");
 XPRBfreemem(iisvar);          /* Free the array of variables */
 printf("Constraints: ");      /* Print all constraints */
 for(i=0;i<numc;i++) printf("%s ", XPRBgetctrname(iisctr[i]));
 printf("\n");
 XPRBfreemem(iisctr);          /* Free the array of constraints */
 printf("SOS: ");              /* Print all SOS */
 for(i=0;i<nums;i++) printf("%s ", XPRBgetsosname(iissos[i]));
 printf("\n");
 XPRBfreemem(iissos);          /* Free the array of SOS */
}
```

Further information

1. This function returns the variables, constraints, and SOS forming an IIS (irreducible infeasible set) in an infeasible MIP problem. The number of independent IIS identified by Xpress Optimizer can be obtained with function `XPRBgetnumiis`.
2. The arrays that are allocated by this function must be freed by the user's program by calls to `XPRBfreemem`.
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, constraints, and SOS.

Related topics

`XPRBgetnumiis`, `XPRBgetmipstat`, `XPRBgetiis`.

XPRBgetmipstat

Purpose

Get the MIP status.

Synopsis

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

Argument

`prob`	Reference to a problem.

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.
`XPRB_MIP_UNBOUNDED`	LP unbounded;

Example

The following returns the current MIP status.

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

Further information

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

Related topics

`XPRBgetlpstat`, `XPRBgetprobstat`.

XPRBgetnextterm

Purpose

Get the next linear term of a constraint.

Synopsis

```
const void *XPRBgetnextterm(XPRBctr ctr, const void *ref, XPRBvar *var,
      double *coeff);
```

Arguments

`ctr`	Constraint whose terms are to be enumerated.
`ref`	Reference pointer or `NULL`.
`var`	BCL reference to a variable. May be `NULL` if not required.
`coeff`	Coefficient associated to variable `var`. May be `NULL` if not required.

Return value

Reference pointer for the next call to `XPRBgetnextterm` or `NULL` if there are no more terms.

Example

```
XPRBprob prob;
XPRBctr ctr;
XPRBvar var;
double coeff;
const void *ref;
   ...
ref = NULL;
while ((ref = XPRBgetnextterm(ctr, ref, &var, &coeff)) != NULL) {
  XPRBprintf("Variable %s has coefficient %g\n",
    XPRBgetvarname(var), coeff);
}
```

This code extract prints all linear terms of a constraint.

Further information

This function can be used to enumerate the linear terms of a constraint. The second parameter serves to keep track of current location in the enumeration; if this parameter is `NULL`, the first term is returned. This function returns `NULL` if it is called with the reference to the last element. Otherwise, the returned value can be used as the input parameter `ref` to retrieve the following term of the same type. Note that the constant term of the constraint is not included in this enumeration (it can be retrieved with `XPRBgetcoeff`)

Related topics

`XPRBgetnextqterm`, `XPRBaddterm`.

XPRBgetnextqterm

Purpose

Get the next quadratic term of a constraint.

Synopsis

```
const void *XPRBgetnextqterm(XPRBctr ctr, const void *ref, XPRBvar *var1,
       XPRBvar *var2, double *coeff);
```

Arguments

`ctr`	Constraint whose terms are to be enumerated.
`ref`	Reference pointer or `NULL`.
`var1`	BCL reference to a variable. May be `NULL` if not required.
`var2`	BCL reference to a variable. May be `NULL` if not required.
`coeff`	Coefficient associated to the quadratic term `var1 * var2`. May be `NULL` if not required.

Return value

Reference pointer for the next call to `XPRBgetnextqterm` or `NULL` if there are no more terms.

Example

```
XPRBprob prob;
XPRBctr ctr;
XPRBvar var1, var2;
double coeff;
const void *ref;
   ...
ref = NULL;
while ((ref = XPRBgetnextqterm(ctr, ref, &var1, &var2, &coeff))
       != NULL) {
  XPRBprintf("Quadratic term %s * %s has coefficient %g\n",
    XPRBgetvarname(var1), XPRBgetvarname(var2), coeff);
}
```

This example prints all quadratic terms of a constraint.

Further information

This function can be used to enumerate the quadratic terms of a constraint. The second parameter serves to keep track of the current location inthe enumeration; if this parameter is `NULL`, the first term is returned. This function returns `NULL` if it is called with the reference to the last element. Otherwise, the returned value can be used as the input parameter `ref` to retrieve the following term of the same type.

Related topics

`XPRBgetnextterm`, `XPRBaddqterm`.

XPRBgetnextctr

Purpose

Get the next constraint defined in the problem.

Synopsis

```
XPRBctr XPRBgetnextctr(XPRBprob prob, XPRBctr ref);
```

Arguments

`prob`	Reference to a problem.
`ref`	Reference constraint or NULL.

Return value

The next constraint in the enumeration order, or `NULL` .

Example

```
XPRBprob prob;
XPRBctr ctr;
    ...
ctr = NULL;
while ((ctr = XPRBgetnextctr(prob, ctr)) != NULL) {
  XPRBprintctr(ctr);
}
```

This prints all constraints defined in the problem.

Further information

This function can be used to enumerate the constraints of a problem. The second parameter serves to keep track of the current location in the enumeration; if this parameter is `NULL`, the first constraint is returned, otherwise the constraint that follows it is returned. This function returns `NULL` if `ref` is the last constraint (or if there are no constraints to enumerate).

Related topics

`XPRBnewctr`, `XPRBgetnextterm`.

XPRBgetmodcut

Purpose

Get the *model cut* 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 ctr1;
int mcstat;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
mcstat = XPRBgetmodcut(ctr1);
```

This determines whether `ctr1` 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");
   ...
XPRBlpoptimize(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 or MIP 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");
 ...
XPRBlpoptimize(expl2, "");
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`, `XPRBsetprobname`.

XPRBgetprobstat

Purpose

Get the problem status.

Synopsis

```
int XPRBgetprobstat(XPRBprob prob);
```

Argument

`prob`	Reference to a problem.

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.

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 = XPRBgetprobstat(expl2);
if((status&XPRB_MOD)==XPRB_MOD)
  XPRBlpoptimize(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`.

XPRBgetqcoeff

Purpose

Get the coefficient of a quadratic constraint term.

Synopsis

```
double XPRBgetqcoeff(XPRBctr ctr, XPRBvar var1, XPRBvar var2);
```

Arguments

`ctr`	BCL reference to a previously created constraint.
`var1`	BCL reference to a variable.
`var2`	BCL reference to a variable (not necessarily different from first variable).

Return value

Coefficient of the variable in the specified constraint or 0 if the variable does not occur.

Example

```
XPRBprob expl2;
XPRBctr ctr1;
XPRBvar x1;
double val;
   ...
expl2 = XPRBnewprob("example2");
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 0, 100);
ctr1 = XPRBnewctr(expl2, "r1", XPRB_E);
XPRBaddqterm(ctr1, x1, x1, 5.4);
val = XPRBgetqcoeff(ctr1, x1, x1);
```

This retrieves the coefficient of the quadratic term `x1*x1` in the constraint `ctr1`.

Further information

This function returns the coefficient of a given quadratic term `var1*var2` in the constraint `ctr`. Return value 0 indicates that the term is not contained in the constraint.

Related topics

`XPRBaddqterm`, `XPRBgetcoeff`, `XPRBgetrhs`, `XPRBnewctr`, `XPRBsetqterm`.

XPRBgetrange

Purpose

Get the range values for a range constraint.

Synopsis

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

Arguments

`ctr`	Reference to a range constraint.
`bdl`	Lower bound on the range constraint. May be `NULL` if not required.
`bdu`	Upper bound on the range constraint. May be `NULL` 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);
XPRBlpoptimize(expl2, "");
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.
Reduced cost information is available only after LP solving. To obtain reduced cost values for a MIP solution (that is, if function `XPRBgetmipstat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), you need to fix the discrete variables to their solution values with a call to `XPRSfixglobals`, followed by a call to `XPRBlpoptimize` before calling `XPRBgetrcost`. Otherwise, if this function is called when a MIP solution is available it returns 0.
If no solution information is available this function outputs a warning and returns 0.
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`. In this case it returns the reduced cost value in the last LP that has been solved.

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`, `XPRBgetcoeff`, `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
`XPRB_MAXIM`	the objective function is to be maximized;
`XPRB_MINIM`	the objective function is to be minimized;
`-1`	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 function `XPRBsetsense`.

Related topics
`XPRBlpoptimize`, `XPRBmipoptimize`, `XPRBsetsense`.

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);
XPRBlpoptimize(expl2, "");
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);
XPRBlpoptimize(expl2, "");
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`.

XPRBgetsolvar

Purpose

Get the value assigned to a variable in a solution.

Synopsis

```
int XPRBgetsolvar(XPRBsol sol, XPRBvar var, double *val);
```

Arguments

`sol`	Reference to a previously created solution.
`var`	Reference to a previously created variable.
`val`	Pointer to a double where the value will be returned.

Return value

0	variable `var` is assigned a value in the solution and the value is returned in `val`;
−1	variable `var` is not assigned any value in the solution (`val` is left unmodified);
1	an error has occurred.

Example

The following example retrieves the value assigned to variable `x1` in the solution `sol1`.

```
XPRBprob expl2;
XPRBsol sol1;
XPRBvar x1;
double val;
   ...
expl2 = XPRBnewprob("example2");
x1 = XPRBnewvar(expl2, XPRB_UI, "abc3", 0, 100);
sol1 = XPRBnewsol(expl2);
XPRBsetsolvar(sol1, x1, 5.4);
XPRBgetsolvar(sol1, x1, &val);
```

Related topics

`XPRBdelsolvar`, `XPRBnewsol`, `XPRBsetsolvar`, `XPRBsetsolarrvar`.

XPRBgetsolsize

Purpose

Get the size of an `XPRBsol` solution.

Synopsis

```
int XPRBgetsolsize(XPRBsol sol);
```

Argument

`sol` Reference to a previously created solution.

Return value

Size (= number of variables that have been assigned a value) of the solution, or -1 in case of an error.

Example

The following returns the size of the solution `sol1`.

```
XPRBprob expl2;
XPRBsol sol1;
int size;
   ...
expl2 = XPRBnewprob("example2");
sol1 = XPRBnewsol(expl2);
   ...
size = XPRBgetsolsize(sol1);
```

Related topics

`XPRBdelsolvar`, `XPRBnewsol`, `XPRBsetsolvar`, `XPRBsetsolarrvar`.

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`	BCL reference to a variable.
`rngtype`	The type of ranging information sought. This is one of:

`XPRB_UPACT`	upper activity (= the activity level [solution value] that would result from a cost coefficient increase from the input cost to the upper cost XPRB_UCOST—assuming a minimization problem—ignoring the upper bound on this variable);
`XPRB_LOACT`	lower activity (= the activity level [solution value] that would result from a cost coefficient decrease from the input cost to the lower cost XPRB_LCOST—assuming a minimization problem—ignoring the upper bound on this variable);
`XPRB_UUP`	upper unit cost (= the change in the objective function per unit of change in the activity up to the upper activity XPRB_UPACT);
`XPRB_UDN`	lower unit cost (= the change in the objective function per unit of change in the activity down to the lower activity XPRB_LOACT)
`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);
 ...
XPRBlpoptimize("expl2, "");
ucval = XPRBgetvarrng(x1, XPRB_UCOST);
```

Further information

1. 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 (using `XPRSfixglobals`) and re-solving the resulting LP problem.

2. For non-basic variables, the unit costs are always the (absolute) values of the reduced costs.

Related topics

`XPRBnewvar`, `XPRBgetctrrng`.

XPRBgetvartype

Purpose

Get the type of a variable.

Synopsis

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

Argument

`var`	BCL reference to a variable.

Return value

`XPRB_PL`	continuous;
`XPRB_BV`	binary;
`XPRB_UI`	general integer;
`XPRB_PI`	partial integer;
`XPRB_SC`	semi-continuous;
`XPRB_SI`	semi-continuous integer;
`-1`	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 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. With BCL C, the initialization is also performed by function `XPRBnewprob` so that it is not required to call this explicit initialization, particularly for stand-alone model runs. 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.
3. In applications that create a large number of problems it is recommended to use the explicit initialization—once only per process for highest efficiency.
4. 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");
XPRBlpoptimize(expl2, "");
basis = XPRBsavebasis(expl2);
   ...
XPRBloadmat(expl2);
XPRBloadbasis(basis);
XPRBdelbasis(basis);
XPRBlpoptimize(expl2, "");
```

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 since the basis has been stored (addition of variables and constraints is handled—deletion of constraints is not handled: instead of entirely deleting a constraint, change its type to `XPRB_N` using `XPRBsetctrtype` if you wish to load the basis later on). 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), `XPRSwritebasis` (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`, `XPRBgetXPRSprob`, Appendix B.

XPRBloadmipsol

Purpose

Load an integer solution into BCL or the Optimizer.

Synopsis

```
int XPRBloadmipsol(XPRBprob prob, double *sol, int ncol, int ifopt);
```

Arguments

`prob`	Reference to a problem.
`sol`	Array of size `ncol` holding the solution values.
`ncol`	Number of variables (continuous+discrete) in the problem.
`ifopt`	Whether to load the solution into the Optimizer: `0` load into BCL only; `1` load solution into the Optimizer.

Return value

`0`	solution accepted,
`1`	solution rejected because it is infeasible,
`2`	solution rejected because it is cut off,
`3`	solution rejected because the LP reoptimization was interrupted,
`-1`	solution rejected because an error occurred,
`-2`	the given solution array does not have the expected size,
`-3`	error loading solution into BCL.

Example

Load a MIP solution for problem `expl2` into BCL, but not into the Optimizer. We know that the problem has 5 variables.

```
XPRBprob expl2;
double vals[] = {1.5, 1, 0, 4, 2.2};
expl2 = XPRBnewprob("example2");
   ...
if (XPRBloadmipsol(expl2, vals, 5, 0)!=0)
 printf("Loading the solution failed.\n");
```

Further information

1. This function loads a MIP solution from an external source (*e.g.*, the Xpress MIP Solution Pool) into BCL or the Optimizer. The solution is given in the form of an array, indexed by the column numbers of the decision variables. The size `ncol` of the array must correspond to the number of columns in the matrix (generated by a call to `XPRBloadmat` or by starting an optimization run from BCL). If the solution is loaded into BCL the values are accepted as is, if the solution is loaded into the Optimizer (`ifopt` = 1), the Optimizer will check whether the solution is acceptable and recalculates the values for the continuous variables in the solution. In the latter case the solution is loaded into BCL only once it has been successfully loaded and validated by the Optimizer.

2. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition, it is regenerated automatically before loading the solution.

Related topics

`XPRBaddmipsol`, `XPRBgetcolnum`, `XPRBloadmat`, `XPRBgetobjval`, `XPRBgetsol`.

XPRBlpoptimize

Purpose

Solve as a continuous problem.

Synopsis

```
int XPRBlpoptimize(XPRBprob prob, char *alg);
```

Arguments

`prob` Reference to a problem.

`alg` Choice of the solution algorithm and options, as a string of flags. If no flag is specified, solve the problem using the default LP/QP algorithm; otherwise, if the argument includes:

- `"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;
- `"c"` continue a previously interrupted optimization run;

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code uses the primal simplex algorithm to solve `expl2` as a continuous problem.

```
XPRBprob expl2;
expl2 = XPRBnewprob("example2");
  ...
XPRBlpoptimize(expl2, "p");
```

Further information

This function selects and starts the Xpress Optimizer LP/QP solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, *e.g.* "`pn`. 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`.

Related topics

`XPRBgetsense`, `XPRBlpoptimize`, `XPRBsetobj`, `XPRBsetsense`, `XPRBsync`.

XPRBmipoptimize

Purpose

Solve as a discrete problem.

Synopsis

```
int XPRBmipoptimize(XPRBprob prob, char *alg);
```

Arguments

`prob` Reference to a problem.

`alg` Choice of the solution algorithm and options, as a string of flags. If no flag is specified, solve the problem using the default MIP/MIQP algorithm; otherwise, if the argument includes:

- `"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 (for the initial LP);
- `"l"` stop after solving the initial continuous relaxation (using MIP information in presolve);
- `"c"` continue a previously interrupted optimization run;

Return value

0 if function executed successfully, 1 otherwise.

Example

The following code uses the default algorithm to solve `expl2` as a discrete problem.

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

Further information

This function selects and starts the Xpress Optimizer MIP/MIQP solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, *e.g.* "`pn`. 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 `XPRSpostsolve` in order to remove all search tree information that has been stored.

Related topics

`XPRBgetsense`, `XPRBlpoptimize`, `XPRBsetobj`, `XPRBsetsense`, `XPRBsync`.

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 `NULL` if not required.
`av`	Reference to an array of variables.
`cof`	Array of constant coefficients for all elements of `av`. It must be at least the same size as `av`.
`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`.)

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

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 `NULL` 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 `NULL` if not required.
`type`	Type of the constraint, which must be one of `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).

Return value

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

Example

The following creates a new equality constraint.

```
XPRBprob prob;
XPRBctr ctr1;
 ...
ctr1 = 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

`prob`	Reference to a problem.
`qrtype`	Type of the cut: `XPRB_L` $\leq$ (inequality) `XPRB_G` $\geq$ (inequality) `XPRB_E` = (equation)
`mtype`	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 `av`).
`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, "arry1", 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`.

Note: all terms that are added to a cut must belong to the same problem as the cut itself.

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`.

Note: all terms that are added to a cut must belong to the same problem as the cut itself.

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`.

Note: all terms that are added to a cut must belong to the same problem as the cut itself.

Related topics

`XPRBnewcut`, `XPRBaddcutterm`.

XPRBnewidxset

Purpose
Create a new index set.

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

Arguments
`prob`	Reference to a problem.
`name`	Name of the index set to be created. May be `NULL` if not required.
`maxsize`	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

`format` String indicating the printing format using standard C conventions (see the documentation of `printf` in a C manual for a complete list of format options). Simple formating options are of the form %*n* where *n* may be, for instance, one of

- `c` single character;
- `d` integer;
- `g` double;
- `s` string of characters.

`...` 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 `NULL` 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`.)

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

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 above remarks on initialization of BCL through this function apply when using BCL in C, initialization for other interfaces may differ—please refer to the specific interface documentation.
3. 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`.

XPRBnewsol

Purpose

Create a solution.

Synopsis

```
XPRBsol XPRBnewsol(XPRBprob prob);
```

Argument

`prob` Reference to a problem.

Return value

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

Example

The following creates a solution sol1.

```
XPRBprob prob;
XPRBsol sol1;
    ...
sol1 = XPRBnewsol(prob);
```

Further information

This function creates an `XPRBsol` solution. It returns the address of the solution that is taken as a parameter in the functions for adding variables, such as `XPRBsetsolvar`, deleting variables `XPRBdelsolvar` or the entire solution `XPRBdelsol`. Note that `XPRBsol` solutions represent user-defined solutions to be passed to the Optimizer, not solutions retrieved from the Optimizer.

Related topics

`XPRBdelsol`, `XPRBdelsolvar`, `XPRBgetsolsize`, `XPRBgetsolvar`, `XPRBprintsol`, `XPRBsetsolarrvar`, `XPRBsetsolvar`.

XPRBnewsos

Purpose

Create a SOS.

Synopsis

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

Arguments

`prob`	Reference to a problem.
`name`	The name of the set.
`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.

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

`prob` Reference to a problem.

`name` Name of the set.

`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` Array of variables. May be `NULL` if not required.

`ctr` Reference to a constraint which has been previously defined. May be `NULL` of 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`.)

Note: all members that are added to a SOS must belong to the same problem as the SOS itself.

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 `NULL` 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`.)

Note: all members that are added to a SOS must belong to the same problem as the SOS itself.

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 `NULL` 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`.)

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

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 `NULL` 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 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, "arry1", 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 *cut*2.

```
XPRBcut cut2;
XPRBarrvar ty1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
ty1 = XPRBnewarrvar(epl1, 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

`prob`	Reference to a problem.
`format`	String indicating the format of the text to be output. Format parameters are identical to those of the C function `printf`.
`...`	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`.

XPRBprintsol

Purpose

Print out a solution.

Synopsis

```
int XPRBprintsol(XPRBsol sol);
```

Argument

`sol` Reference to a solution.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;
XPRBsol sol1;
   ...
sol1 = XPRBnewsol(prob);
   ...
XPRBprintsol(sol1);
```

This prints out the solution `sol1`.

Further information

This function prints out an `XPRBsol` solution (note that `XPRBsol` solutions represent user-defined solutions to be passed to the Optimizer, not solutions coming from the Optimizer). A solution is printed as a sequence like "*varname* = *value*, ... ". If the solution doesn't contain any variable, only an empty line is printed. This function is not available in the student version.

Related topics

`XPRBnewsol`, `XPRBprintctr`, `XPRBprintidxset`, `XPRBprintprob`, `XPRBprintvar`.

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

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

XPRBreadbinsol

Purpose

Read a solution from a binary solution, loading it into the Optimizer.

Synopsis

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

Arguments

`prob`	Reference to a problem.
`fname`	Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .sol will be appended.
`flags`	Flags to control solution import. If no flags need to be specified, use either `NULL` or the empty string. Refer to function `XPRSreadbinsol` in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if function executed successfully, 1 otherwise.

Example

This example reads a solution from file `example2.sol`.

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

Related topics

`XPRBreadslxsol`, `XPRBwritebinsol`.

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

`fgs` The system's `fgets` function (defined by `XPRB_FGETS`).

`file` Pointer to a data file.

`length` Maximum length of any text string to be read.

`format` String indicating the format of the data line to be read, which may be one of:

- `t[num`] text up to next separator sign or space (blank / tab / line break), optionally followed by the maximum string length to be read;
- `s[num`] text marked by single quotes, `' '`, optionally followed by the maximum string length to be read;
- `S[num`] text marked by double quotes, `" "`, optionally followed by the maximum string length to be read;
- `T[num`] text as for `t`, `s`, or `S`, depending on the first character read, optionally followed by the maximum string length to be read;
- `i` integer value;
- `g` real (float) value.

The number of format parameters is arbitrary.

`...` 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.

XPRBreadslxsol

Purpose

Read a solution from an ASCII solution file (.slx), loading it into the Optimizer.

Synopsis

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

Arguments

`prob` Reference to a problem.

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .slx will be appended.

`flags` Flags to control solution import. If no flags need to be specified, use either `NULL` or the empty string. Refer to function `XPRSreadslxsol` in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if function executed successfully, 1 otherwise.

Example

This example reads a solution from file `example2.slx`.

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

Related topics

`XPRBreadbinsol`, `XPRBwriteslxsol`.

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):
 ...
XPRBlpoptimize(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 spare 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");
 ...
XPRBlpoptimize(expl2, "");
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

av	BCL reference to an array.
ndx	Index within the array.
var	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.

Note: all variables that are added to an array of variables must belong to the same problem as the array itself.

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

`ctr`	Reference to a previously created constraint.
`qrtype`	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 function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;
XPRBctr ctr1;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBsetctrtype(ctr1, XPRB_L);
```

This changes `ctr1` 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

`cut` Reference to a previously created cut.

`id` Classification or identification number.

Return value

0 if function executed successfully, 1 otherwise.

Example

Set the classification or identification number of the cut *cut*1 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 *cut*1 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`.

Note: all terms that are added to a cut must belong to the same problem as the cut itself.

Related topics

`XPRBnewcut`, `XPRBaddcutterm`, `XPRBdelcutterm`.

XPRBsetcuttype

Purpose

Set the type of a cut.

Synopsis

```
int XPRBsetcuttype(XPRBcut cut, int type);
```

Arguments

`cut`	Reference to a previously created cut.
`type`	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 *cut*1 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`.

XPRBsetdelayed

Purpose

Set the constraint type.

Synopsis

```
int XPRBsetdelayed(XPRBctr ctr, int dstat);
```

Arguments

`ctr`	Reference to a previously created constraint.
`dstat`	The constraint type, which must be one of: `0` ordinary constraint; `1` delayed constraint.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following turns the constraint `ctr1` into a delayed costraint.

```
XPRBprob prob;
XPRBctr ctr1;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBsetdelayed(ctr1, 1);
```

Further information

1. This function changes the type of a previously defined constraint from ordinary constraint to delayed constraint and vice versa.
2. Delayed or 'lazy' constraints must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.
3. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Related topics

`XPRBgetdelayed`, `XPRBnewctr`, `XPRBsetincvars`, `XPRBsetindicator`, `XPRBsetmodcut`.

XPRBsetdictionarysize

Purpose

Set the size of a dictionary.

Synopsis

```
int XPRBsetdictionarysize(XPRBprob prob, int dict, int size)
```

Arguments

`prob`	Reference to a problem.
`dict`	Choice of the dictionary. Possible values: `XPRB_DICT_NAMES` names dictionary `XPRB_DICT_IDX` indices dictionary
`size`	Non-negative value, preferrably 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");
XPRBsetdictionarysize(expl2, XPRB_DICT_NAMES, 0);
```

Further information

1. This function sets the size of the hash table of the names or indices dictionaries 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 this dictionary is enabled (default setting) BCL automatically resizes this dictionary to a suitable size for your problem. If nevertheless you wish to set the size by yourself we recommend to choose a value close to the number of variables+constraints in your problem.
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`.

XPRBsetincvars

Purpose
Set the *include vars* constraint type.

Synopsis
```
int XPRBsetincvars(XPRBctr ctr, int ivstat);
```

Arguments
`ctr` Reference to a previously created constraint.

`ivstat` The constraint type, which must be one of:
- `0` normal constraint;
- `1` *include vars* special constraint.

Return value
0 if function executed successfully, 1 otherwise.

Example
The following turns the constraint `ctr1` into an *include vars* special constraint.

```
XPRBprob prob;
XPRBctr ctr1;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_N);
XPRBsetincvars(ctr1, 1);
```

Further information
1. This function changes the type of a previously defined constraint from ordinary constraint to an *include vars* special constraint. *Include vars* constraints are used to force the loading of all variables they contain into the Optimizer (even if they don't appear in any other constraint). Only constraints of type `XPRB_N` can be changed into *include vars* constraints; the constraints themselves are not loaded into the Optimizer (as all constraints of type `XPRB_N`), just the variables they contain are loaded. The coefficients of the variables are also ignored as long as they are non-zero.

2. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Related topics
`XPRBgetincvars`, `XPRBnewctr`, `XPRBsetdelayed`, `XPRBsetindicator`, `XPRBsetmodcut`.

XPRBsetindicator

Purpose

Set the indicator constraint type.

Synopsis

```
int XPRBsetindicator(XPRBctr ctr, int dir, XPRBvar b);
```

Arguments

`ctr` Reference to a previously created inequality or range constraint.

`dstat` The indicator type, which must be one of:
- `-1` indicator constraint with condition $b = 0$;
- `0` ordinary constraint;
- `1` indicator constraint with condition $b = 1$.

`b` Reference to a previously created binary variable.

Return value

0 if function executed successfully, 1 otherwise.

Example

The following turns the constraint `ctr1` into the indicator constraint $b = 1 \Rightarrow ctr1$.

```
XPRBprob prob;
XPRBctr ctr1;
XPRBvar b;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_L);
b = XPRBnewvar(prob, XPRB_BV, "b", 0, 1);
XPRBsetindicator(ctr1, 1, b);
```

Further information

1. This function changes the type of a previously defined constraint from ordinary constraint to indicator constraint and vice versa.

2. Indicator constraints are defined by associating a binary variable and an implication sense with a linear inequality or range constraint.

3. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Related topics

`XPRBgetindicator`, `XPRBgetindvar`, `XPRBnewctr`, `XPRBsetincvars`, `XPRBsetdelayed`, `XPRBsetmodcut`.

XPRBsetlb

Purpose

Set a lower bound.

Synopsis

```
int XPRBsetlb(XPRBvar var, double bdl);
```

Arguments

`var`	BCL reference to a variable.
`bdl`	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

`var`	BCL reference to a variable.
`c`	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
`ctr` Reference to a previously created constraint.

`mcstat` 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 `ctr1` into a model cut.

```
XPRBprob prob;
XPRBctr ctr1;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBsetmodcut(ctr1, 1);
```

Further information
1. This function changes the type of a previously defined constraint from ordinary constraint to model cut and vice versa.
2. Model cuts must be 'true' cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.
3. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Related topics
`XPRBgetmodcut`, `XPRBnewctr`, `XPRBsetincvars`, `XPRBsetdelayed`, `XPRBsetindicator`.

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.e.* 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`.

XPRBsetprobname

Purpose

Set the name of the specified problem.

Synopsis

```
int XPRBsetprobname(XPRBprob prob, const char *name);
```

Arguments

`prob`	Reference to a problem.
`name`	A string of up to 1024 characters containing the new problem name.

Return value

0 if function executed successfully, 1 otherwise.

Example

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

This creates a new problem, than changes its name and prints the new name `example_two`.

Related topics

`XPRBdelprob`, `XPRBgetprobname`, `XPRBnewname`, `XPRBnewprob`.

XPRBsetqterm

Purpose

Set a quadratic constraint term.

Synopsis

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

Arguments

`ctr`	Reference to a previously defined constraint.
`var1`	Reference to a variable.
`var2`	Reference to a variable (not necessarily different from first variable).
`coeff`	Value to be added to the coefficient of the term `var1 * var2`.

Return value

0 if function executed successfully, 1 otherwise.

Example

```
XPRBprob prob;
XPRBvar x2;
XPRBctr ctr1;
 ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_L);
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
XPRBaddqterm(ctr1, x2, x2, 1);
XPRBsetqterm(ctr1, 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`.

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

Related topics

`XPRBaddqterm`, `XPRBdelqterm`.

XPRBsetrange

Purpose

Define a range constraint.

Synopsis

```
int XPRBsetrange(XPRBctr ctr, double bdl, double bdu);
```

Arguments

`ctr`	Reference to the constraint.
`bdl`	Lower bound on the range constraint.
`bdu`	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

`prob` Reference to a problem.

`fmt` Format string (as used by the C function `printf`). Simple format strings are of the form `%`*n* where *n* may be, for instance, one of

- `g` 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)
- `.numf` print real numbers in the style [-]d.d where the number of digits after the decimal point is equal to the given precision `num`.

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

`prob`	Reference to a problem.
`dir`	Sense of the objective function, which must be one of: `XPRB_MAXIM` maximize the objective; `XPRB_MINIM` 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

`sos`	Reference to a previously created SOS.	
`type`	The directive type, which must be one of:	
	`XPRB_PR`	priority;
	`XPRB_UP`	first branch upwards;
	`XPRB_DN`	first branch downwards;
	`XPRB_PU`	pseudo cost on branching upwards;
	`XPRB_PD`	pseudo cost on branching downwards.
`val`	An argument dependent on the type of the directive being defined. If `type` is:	
	`XPRB_PR`	`val` will be the priority value, an integer between `1` (highest) and `1000` (lowest), the default;
	`XPRB_UP`	no input is required — choose any value, *e.g.* `0`;
	`XPRB_DN`	no input is required — choose any value, *e.g.* `0`;
	`XPRB_PU`	`val` will be the value of the pseudo cost for the upward branch;
	`XPRB_PD`	`val` 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. 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`.

XPRBsetsolarrvar

Purpose
Set the values assigned to multiple variables in a solution.

Synopsis
```
int XPRBsetsolarrvar(XPRBsol sol, const XPRBarrvar av, const double val[]);
```

Arguments
`sol`	BCL reference to a previously created solution.
`av`	Reference to an array of variables.
`val`	Values to be assigned to 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
```
XPRBprob prob;
XPRBsol sol1;
XPRBarrvar ty1;
double cr[] = {2, 13, 15, 6, 8.5};
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
   ...
sol1 = XPRBnewsol(prob);
XPRBsetsolarrvar(sol1, ty1, cr);
```

Further information
This function sets multiple variables to the given values in a solution, the variables coming from array `av` and the corresponding values from `val`. If a variable was already assigned a value in that solution, the value is overwritten.

Note: all variables that are added to a solution must belong to the same problem as the solution itself.

Related topics
XPRBdelsolvar, XPRBnewsol, XPRBsetsolvar.

XPRBsetsolvar

Purpose
Set the value assigned to a variable in a solution.

Synopsis
```
int XPRBsetsolvar(XPRBsol sol, const XPRBvar var, double val);
```

Arguments
`sol`	BCL reference to a previously created solution.
`var`	BCL reference to a variable.
`val`	Value assigned to the variable `var`.

Return value
0 if function executed successfully, 1 otherwise.

Example
```
XPRBprob prob;
XPRBsol sol1;
XPRBvar x1;
x1 = XPRBnewvar(expl1, XPRB_UI, "abc3", 0, 100);
   ...
sol1 = XPRBnewsol(prob);
XPRBsetsolvar(sol1, x1, 7.0);
```

This example sets variable `x1` to value `7.0` in solution `sol1`.

Further information
This function sets a variable to the given value in a solution. If the variable was already assigned a value in that solution, the value is overwritten.

Note: all variables that are added to a solution must belong to the same problem as the solution itself.

Related topics
`XPRBdelsolvar`, `XPRBnewsol`, `XPRBsetsolarrvar`.

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 ctr1;
   ...
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBsetterm(ctr1, NULL, 7.0);
```

This sets the right hand side of the constraint `ctr1` 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`.

Note: all terms that are added to a constraint must belong to the same problem as the constraint itself.

Related topics

`XPRBaddterm`, `XPRBdelctr`, `XPRBnewctr`.

XPRBsetub

Purpose

Set an upper bound.

Synopsis

```
int XPRBsetub(XPRBvar var, double bdu);
```

Arguments

`var` BCL reference to a variable.

`bdu` 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

`var` BCL reference to a variable.

`type` Directive type, which must be one of:

`XPRB_PR`	priority;
`XPRB_UP`	first branch upwards;
`XPRB_DN`	first branch downwards;
`XPRB_PU`	pseudo cost on branching upwards;
`XPRB_PD`	pseudo cost on branching downwards.

`c` An argument dependent on the type of directive to be defined. Must be one of:

`XPRB_PR`	priority value — an integer between `1` (highest) and `1000` (least priority), the default;
`XPRB_UP`	no input required — set to any value, *e.g.* `0`;
`XPRB_DN`	no input required — set to any value, *e.g.* `0`;
`XPRB_PU`	value of the pseudo cost on branching upwards;
`XPRB_PD`	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. 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 possibly 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
`var`	Reference to a BCL variable
`link`	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

`var`	BCL reference to a variable.
`type`	The variable type, which is 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.

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`.

XPRBstartarrvar

Purpose

Start the definition of a variable array.

Synopsis

```
XPRBarrvar XPRBstartarrvar(XPRBprob prob, int nbvar, const char *name);
```

Arguments

`prob`	Reference to a problem.
`nbvar`	The maximum number of variables in the array.
`name`	Name of the array. May be `NULL` 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 LP solution currently held in the Optimizer;
	`XPRB_XPRS_SOLMIP`	update the BCL solution information with the last MIP solution found by 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");
 ...
XPRBmipoptimize(expl2, "");
 ...
XPRBsync(expl2, XPRB_XPRS_PROB);
XPRBmipoptimize(expl2, "");
```

Further information

1. This function resets the BCL problem status.
2. `XPRB_XPRS_SOL`: retrieves the current LP solution (through `XPRSgetlpsol` function and `XPRS_LPOBJVAL` attribute); correctly used also in *intsol* callbacks as, when an integer solution is found during a global search, it is always set up as an LP solution to the current node.
3. `XPRB_XPRS_SOLMIP`: retrieves the last MIP solution found (through `XPRSgetmipsol` function and `XPRS_MIPOBJVAL` attribute); if used from an *intsol* callback, it will not necessarily return the solution that caused the invocation of the callback (it is possible that another thread finds a new integer solution before that one is retrieved).
4. `XPRB_XPRS_SOL` and `XPRB_XPRS_SOLMIP`: the solution information in BCL is updated with the solution held in the Optimizer at the next solution access (only the objective value is updated immediately).
5. `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`, `XPRBlpoptimize`, `XPRBmipoptimize`.

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`.

XPRBwritesol

Purpose

Write the current Optimizer solution to a CSV format ASCII file, problem_name.asc (and .hdr).

Synopsis

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

Arguments

`prob` Reference to a problem.

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If no file extension is specified, then extensions .hdr and .asc will be appended.

`flags` Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function `XPRSwritesol` in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if function executed successfully, 1 otherwise.

Example

This example writes the current solution to the file `example2.asc` (and `.hdr`).

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

Further information

This function writes out to a file the current Optimizer solution. If no file extension is specified, then two files will be written with extensions .asc and .hdr 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

`XPRBwritebinsol`, `XPRBwriteprtsol`, `XPRBwriteslxsol`.

XPRBwritebinsol

Purpose
Write the current Optimizer solution to a binary solution file for later input into the Optimizer.

Synopsis
```
int XPRBwritebinsol(XPRBprob prob, const char *fname, const char *flags);
```

Arguments

`prob`	Reference to a problem.
`fname`	Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .sol will be appended.
`flags`	Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function `XPRSwritebinsol` in the 'Xpress Optimizer Reference Manual' for details.

Return value
0 if function executed successfully, 1 otherwise.

Example
This example writes the current solution to the file `example2.sol`.

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

Further information
This function writes out to a file the current Optimizer solution. If no file extension is specified, then the extension .sol 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
`XPRBreadbinsol`, `XPRBwritesol`, `XPRBwriteprtsol`, `XPRBwriteslxsol`.

XPRBwriteprtsol

Purpose

Write the current Optimizer solution to a fixed format ASCII file, problem_name .prt.

Synopsis

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

Arguments

`prob`	Reference to a problem.
`fname`	Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .prt will be appended.
`flags`	Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function `XPRSwriteprtsol` in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if function executed successfully, 1 otherwise.

Example

This example writes the current solution to the file `example2.prt`.

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

Further information

This function writes out to a file the current Optimizer solution. If no file extension is specified, then the extension .prt 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

`XPRBwritesol`, `XPRBwritebinsol`, `XPRBwriteslxsol`.

XPRBwriteslxsol

Purpose

Write the current Optimizer solution to an ASCII solution file (.slx) using a similar format to MPS files.

Synopsis

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

Arguments

`prob` Reference to a problem.

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .slx will be appended.

`flags` Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function `XPRSwriteslxsol` in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if function executed successfully, 1 otherwise.

Example

This example writes the current solution to the file `example2.slx`.

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

Further information

This function writes out to a file the current Optimizer solution. If no file extension is specified, then the extension .slx 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

`XPRBreadslxsol`, `XPRBwritesol`, `XPRBwritebinsol`, `XPRBwriteprtsol`.

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`, `XPRBsol`, `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

 for(j=0;j<NJ;j++)              // Binaries 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_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

 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.mipOptimize();              // 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 3.5.1:

```
#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+= sqr(x[i]-x[j])+sqr(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", sqr(x[i]-CX[i])+sqr(y[i]-CY[i]) <= R[i]);

/****SOLVING + OUTPUT****/
 prob.setSense(XPRB_MINIM);             // Choose sense of optimization
 prob.lpOptimize();                     // 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;
}
```

5.1.3 Error handling

The default behavior of BCL in the case of an error is to output a message and terminate the program. However, in C++ applications it may be more convenient to raise exceptions instead of simply exiting from the program. With the BCL C++ interface the user has the possibility to disable the standard 'exit on error' behavior replacing it, for instance, by C++ exceptions.

The C++ program below implements the example of user error handling from Section 3.6. The default error handling of BCL is disabled (function `XPRBseterrctrl`) and the error handling callback is defined to raise C++ exceptions in the case of an error—the BCL C++ interface uses the callback functions of the BCL C library. When using the BCL C functions with BCL C++ objects we need to employ their C representation (obtained with method `getCRef`).

Besides the user error handling this example also shows how to work with the user message printing callback to redirect the BCL output to a user-defined callback function (this includes output from BCL and anything printed through `XPRBprintf`). By setting the BCL message printing level (method `setMsgLevel`) you can control the amount of information output by BCL.

```
#include <iostream>
#include <string>
#include "xprb_cpp.h"

using namespace std;
using namespace ::dashoptimization;

class bcl_exception
{
 public:
   string msg;
   int code;
   bcl_exception(int c,const char *m)
   {
    code=c;
    msg=string(m);
    cout << "EXCP:" << msg << "\n";
   }
};

/**** User error handling function ****/
void XPRB_CC usererror(xbprob* prob, void *vp, int num, int type,
                       const char *t)
{
 throw bcl_exception(num, t);
}

/**** User printing function ****/
void XPRB_CC userprint(xbprob* 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)
  cout << "BCL output: " << msg;
 else
  cout << msg;
 rtsbefore=(msg[strlen(msg)-1]=='\n');
}

/***********************************************************************/

void modexpl3(XPRBprob &p)
```

```
{
 XPRBvar x[3];
 XPRBlinExp le;
 int i;

 for(i=0;i<2;i++) x[i]=p.newVar(XPRBnewname("x_%d",i), XPRB_UI, 0, 100);

                /* Create the constraints:
                   C1: 2x0 + 3x1 >= 41
                   C2:  x0 + 2x1  = 13 */
 p.newCtr("C1", 2*x[0] + 3*x[1] >= 41);
 p.newCtr("C2", x[0] + 2*x[1] == 13);

// Uncomment the following line to cause an error in the model that
// triggers the user error handling:

// x[2]=p.newVar("x_2", XPRB_UI, 10,1);

 le=0;
 for(i=0;i<2;i++) le += x[i];    // Objective: minimize x0+x1
 p.setObj(le);                   // Select objective function
 p.setSense(XPRB_MINIM);         // Set objective sense to minimization

 p.lpOptimize();                 // Solve the LP
 XPRBprintf(p.getCRef(), "problem status: %d  LP status: %d  MIP status: %d\n",
    p.getProbStat(), p.getLPStat(), p.getMIPStat());

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

 for(i=0;i<2;i++)                // Print solution values
  XPRBprintf(p.getCRef(), "%s:%g, ", x[i].getName(), x[i].getSol());
 XPRBprintf(p.getCRef(), "\n");
}

/*********************************************************************/

int main()
{
 XPRBprob *p;

 XPRBseterrctrl(0);     // Switch to error handling by the user's program
 XPRB::setMsgLevel(2);       // Set the printing flag. Try other values:
                             // 0 - no printed output, 1 - only errors,
                             // 2 - errors and warnings, 3 - all messages
                        // Define the callback functions:
 XPRBdefcbmsg(NULL, userprint, NULL);
 XPRBdefcberr(NULL, usererror, NULL);

 try
 {
  p=new XPRBprob("Expl3");   // Initialize a new problem in BCL
 }
 catch(bcl_exception &e)
 {
  cout << e.code << ":" << e.msg;
  return 1;
 }

 try
 {
  modexpl3(*p);              // Formulate and solve the problem
 }
 catch(bcl_exception &e)
 {
  cout << e.code << ":" << e.msg << "\n";
  return 2;
 }
```

```
    catch(const char *m)
    {
     cout << m << "\n";
     return 3;
    }
    catch(...)
    {
     cout << "other exception\n";
     return 4;
    }
    return 0;
   }
```

5.2 C++ class reference

The complete set of classes of the BCL C++ interface is summarized in the following list:

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, XPRBsol, 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 `XPRBgettime`

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. In applications that create a large number of problems it is recommended to use the explicit initialization—once only per process for highest efficiency.

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.e.* 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

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

```
int setRealFmt(String fmt);
```

Argument

`fmt` Format string (as used by the C function `printf`). Simple format strings are of the form `%`*n* where *n* may be, for instance, one of

- `g` 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)
- `.numf` print real numbers in the style [-]d.d where the number of digits after the decimal point is equal to the given precision `num`.

Return value 0 if method executed successfully, 1 otherwise.

Description

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

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:

```
XPRBprob *prob;

XPRB::init();
XPRB::setRealFmt("%.10f");

prob = new XPRBprob("myprob");
prob->setRealFmt("%g");
```

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

```
XPRBbasis();
XPRBbasis(xbbasis *bs);
```

Argument

`bs` A basis in BCL C.

Description

Create a new basis object.

Method detail

getCRef

Synopsis

```
xbbasis *getCRef();
```

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

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the basis object is correctly defined.

reset

Synopsis

```
void reset();
```

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. Note that all terms in a constraint must belong to the same problem as the constraint itself.

Constructors

`XPRBctr();`

`XPRBctr(xbctr *c);`

`XPRBctr(xbctr *c, XPRBrelation& r);`

Methods

`int 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.

`double getCoefficient(XPRBvar& var);`

`double getCoefficient(XPRBvar& var, XPRBvar& var2);`
Get the coefficient of a constraint term.

`xbctr *getCRef();`
Get the C modeling object.

`double getDual();`
Get dual value.

`int getIndicator();`
Get the indicator type of a constraint.

`XPRBvar getIndVar();`
Get the indicator variable of a constraint.

`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.

`int getSize();`
: Get the size of a constraint.

`double getSlack();`
: Get slack value for a constraint.

`int getType();`
: Get the row type of a constraint.

`bool isDelayed();`
: Check the type of a constraint.

`bool isIncludeVars();`
: Check the type of a constraint.

`bool isIndicator();`
: Check the type of a constraint.

`bool isModCut();`
: Check the type of a constraint.

`bool isValid();`
: Test the validity of the constraint object.

`const void *nextTerm(const void *ref, XPRBvar& var, double *coeff);`

`const void *nextTerm(const void *ref, XPRBvar& var, XPRBvar& v2, double *coeff);`
: Enumerate the terms of a constraint.

`int print();`
: Print out a constraint.

`void reset();`
: Reset the constraint object.

`int setDelayed(bool dstat);`
: Set the constraint type.

`int setIncludeVars(bool ivstat);`
: Set the constraint type.

`int setIndicator(ind dir, XPRBvar );`
: Set the indicator constraint type.

`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

```
XPRBctr();
XPRBctr(xbctr *c);
XPRBctr(xbctr *c, XPRBrelation& r);
```

Arguments

`c` A constraint in BCL C.

`r` Relation defining the constraint.

Description Create a new constraint object.

Method detail

add

Synopsis

```
int add(XPRBexpr& e);
```

Argument `e` A linear or quadratic expression (may be just a single variable or a constant).

Return value 0 if method executed successfully, 1 otherwise.

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 `XPRBctr.setTerm`.

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` or `XPRBaddqterm`

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 solution is available 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 Ctr1;
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 <= 400);

...  // Solve an LP problem

if (Ctr1.getRowNum() >= 0 && prob.getLPStat()==XPRB_LP_OPTIMAL)
{
  cout << Ctr1.getName() << ": activity: " << Ctr1.getAct();
  cout << " = " << Ctr1.getRHS() << " - " << Ctr1.getSlack();
  cout << ", dual: " << Ctr1.getDual() << endl;
}
else
  cout << "No solution information available." << endl;
```

Related topics Calls `XPRBgetact`

getCoefficient

Synopsis

```
double getCoefficient(XPRBvar& var);
double getCoefficient(XPRBvar& var, XPRBvar& var2);
```

Arguments

`var` A BCL variable.

`var2` A second BCL variable (may be the same as `var`).

Return value The coefficient of the given variable or pair of variables, 0 if the constraint does not contain the term.

Description This function returns the coefficient of a given variable `var` or of the quadratic term `var*var2` in the constraint `ctr`. Return value 0 indicates that the term is not contained in the constraint. If `var` is set to `NULL`, this method returns the right hand side (constant term) of the constraint.

Example See `XPRBctr.setTerm`.

Related topics Calls `XPRBgetcoeff` or `XPRBgetqcoeff`

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.
Dual information is available only after LP solving. To obtain dual values for a MIP solution (that is, if function `XPRBprob.getMIPStat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), you need to fix the discrete variables to their solution values with a call to `XPRSfixglobals`, followed by a call to `XPRBlpoptimize` before calling `XPRBgetdual`. Otherwise, if this function is called when a MIP solution is available it returns 0.
If no solution information is available this function outputs a warning and returns 0.
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`. In this case it returns the dual value in the last LP that has been solved.

Example See `XPRBctr.getAct`.

Related topics Calls `XPRBgetdual`

getIndicator

Synopsis

```
int getIndicator();
```

Return value

`0`	an ordinary constraint;
`1`	an indicator constraint with condition $b = 1$;
`-1`	an indicator constraint with condition $b = 0$;
`-2`	an error has occurred.

Description This method returns the indicator status of the given constraint.

Example See `XPRBctr.setIndicator`.

Related topics Calls `XPRBgetindicator`

getIndVar

Synopsis

```
XPRBvar getIndVar();
```

Return value A BCL variable.

Description This method returns the indicator variable associated with the given constraint. This method always returns a BCL variable the validity of which needs to be checked with `XPRBvar.isValid`.

Example See `XPRBctr.setIndicator`.

Related topics Calls `XPRBgetindvar`

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 XPRBctr.setRange.

Related topics Calls XPRBgetrange

getRHS

Synopsis

```
double getRHS();
```

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.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.

Example See XPRBctr.getAct.

Related topics Calls XPRBgetrhs

getRNG

Synopsis

```
double getRNG(int rngtype);
```

Argument

`rngtype` The type of ranging information sought. This is one of:

`XPRB_UPACT`	upper activity (= the level to which the constraint activity may be increased at a cost per unit of increase given by the XPRB_UUP value, ignoring the upper bound on the constraint as specified by its RHS);
`XPRB_LOACT`	lower activity (= the level to which the constraint activity may be decreased at a cost per unit of decrease given by the XPRB_UDN value, ignoring the lower bound on the constraint as specified by its RHS);
`XPRB_UUP`	upper unit cost;
`XPRB_UDN`	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 The following example displays the constraint activity and the activity range.

```
XPRBvar x,y;
XPRBctr Ctr1;
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 <= 400);

...  // Solve the problem

cout << "C1: " << Ctr1.getAct() << " (activity range: ";
cout << Ctr1.getRNG(XPRB_LOACT) << ", " ;
cout << Ctr1.getRNG(XPRB_UPACT) << ")" << endl;
```

Related topics Calls XPRBgetctrrng

getRowNum

Synopsis

```
int getRowNum();
```

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 XPRB_N or no non-zero terms) then the return value is negative. To check whether the matrix has been generated, use method XPRBprob.getProbStat. The counting of row numbers starts with 0.

Example See XPRBctr.getAct.

Related topics Calls XPRBgetrownum

getSize

Synopsis

```
int getSize();
```

Return value Size (= number of linear or quadratic terms with a non-zero coefficient) of the constraint, or -1 in case of an error.

Description This method returns the size of a constraint (or -1 in case of an error).

Example See XPRBctr.setRange.

Related topics Calls XPRBgetctrsize

getSlack

Synopsis

```
double getSlack();
```

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

Description 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.
If this function 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 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 `XPRBprob.sync` with the flag `XPRB_XPRS_SOL`.

Example See `XPRBctr.getAct`.

Related topics Calls `XPRBgetslack`

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);
`XPRB_R`	a range constraint;
`-1`	an error has occurred.

Description This method returns the constraint type if successful, and -1 in case of an error.

Example See `XPRBctr.setRange`.

Related topics Calls `XPRBgetctrtype`

isDelayed

Synopsis

```
bool isDelayed();
```

Return value `true` if constraint is delayed constraint, `false` otherwise.

Description This method indicates whether the given constraint is a delayed or an ordinary constraint.

Related topics Calls `XPRBgetdelayed`

isIncludeVars

Synopsis

```
bool isIncludeVars();
```

Return value `true` if constraint is an *include vars* special constraint, `false` otherwise.

Description This method indicates whether the given constraint is an *include vars*special constraint or an ordinary constraint.

Related topics Calls `XPRBgetincvars`

isIndicator

Synopsis

```
bool isIndicator();
```

Return value `true` if constraint is an indicator constraint, `false` otherwise.

Description This method indicates whether the given constraint is an indicator or an ordinary constraint.

Related topics Calls `XPRBgetindicator`

isModCut

Synopsis

```
bool isModCut();
```

Return value `true` if constraint is a model cut, `false` otherwise.

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

Related topics Calls `XPRBgetmodcut`

isValid

Synopsis

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the constraint object is correctly defined. It should always be used to test the result returned by `XPRBprob.getCtrByName`.

Example See `XPRBprob.getCtrByName`.

nextTerm

Synopsis

```
const void *nextTerm(const void *ref, XPRBvar& var, double *coeff);
const void *nextTerm(const void *ref, XPRBvar& var, XPRBvar& v2, double
        *coeff);
```

Arguments

`ref`	Reference pointer or `NULL`.
`var`	A BCL variable.
`v2`	A second BCL variable (for enumeration of quadratic terms).
`coeff`	Coefficient associated to current term.

Return value Reference pointer for the next call to `nextTerm` or `NULL` if there are no more terms.

Description These methods are used to enumerate the linear or quadratic terms of a constraint. The first parameter `ref` serves to keep track of the current location in the enumeration; if this parameter is `NULL`, the first term is returned. This function returns `NULL` if it is called with the reference to the last element. Otherwise, the returned value can be used as the input parameter `ref` to retrieve the following term of the same type.

Example The following example shows how to enumerate and display the linear terms of a constraint.

```
XPRBprob prob("myprob");
XPRBctr Ctr1;
...
XPRBvar var;
double coeff;
const void *ref = NULL;
while ((ref = Ctr1.nextTerm(ref, var, &coeff)) != NULL) {
  cout << " var " << var.getName() << " has coeff " << coeff << endl;
}
```

Related topics Calls `XPRBgetnextterm` or `XPRBgetnextqterm`

print

Synopsis

```
int print();
```

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 `XPRBctr.setRange`.

Related topics Calls `XPRBprintctr`

reset

Synopsis

```
void reset();
```

Description Clear the definition of the constraint object.

setDelayed

Synopsis

```
int setDelayed(bool dstat);
```

Argument `dstat` The constraint type, which must be one of:

- `false` ordinary constraint;
- `true` delayed constraint.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This method changes the type of a previously defined constraint from ordinary constraint to delayed constraint and vice versa. Delayed or 'lazy' constraints must be satisfied for any integer solution, but will not be loaded into the active set of constraints until required.
2. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Example The following example turns the constraint `Ctr3` into a delayed constraint.

```
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.setDelayed(true);
```

Related topics Calls `XPRBsetdelayed`

setIncludeVars

Synopsis

```
int setIncludeVars(bool ivstat);
```

Argument `ivstat` The constraint type, which must be one of:

- `false` ordinary constraint;
- `true` *include vars* special constraint.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This method changes the type of a previously defined constraint from ordinary constraint to an *include vars* special constraint and vice versa. *Include vars* constraints are used to force the loading of all variables they contain into the Optimizer (even if they don't appear in any other constraint). Only constraints of type `XPRB_N` can be changed into *include vars* constraints; the constraints themselves are not loaded into the Optimizer (as all constraints of type `XPRB_N`), just the variables they contain are loaded. The coefficients of the variables are also ignored as long as they are non-zero.
2. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Example

The following example turns the constraint `CtrIV` into an *include vars* special constraint.

```
XPRBvar y,b;
XPRBctr CtrIV;
XPRBprob prob("myprob");

y = prob.newVar("y", XPRB_PL, 0, 200);
b = prob.newVar("b", XPRB_BV);

CtrIV = prob.newCtr("IncVars", b+y);
CtrIV.setIncludeVars(true);
```

Related topics

Calls `XPRBsetincvars`

---

setIndicator

Synopsis

```
int setIndicator(ind dir, XPRBvar );
```

Arguments

`dir`	The indicator type, which must be one of:	
	`0`	ordinary constraint;
	`-1`	indicator constraint with condition $b = 0$;
	`1`	indicator constraint with condition $b = 1$.
`b`	previously created binary variable.	

Return value

0 if method executed successfully, 1 otherwise.

Description

1. This method changes the type of a previously defined constraint from ordinary constraint to indicator constraint and vice versa. Indicator constraints are defined by associating a binary variable and an implication sense with a linear inequality or range constraint.
2. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Example

The following example turns the constraint `Ctr3` into the indicator constraint $b = 1 \Rightarrow Ctr3$.

```
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);
Ctr3.setIndicator(1, b);
if (Ctr3.isIndicator())
  cout << Ctr3.getIndVar().getName() << "->" << Ctr3.getName() << endl;
```

Related topics Calls XPRBsetindicator

setModCut

Synopsis

```
int setModCut(bool mstat);
```

Argument `mstat` The constraint type, which must be one of:
- `false` constraint;
- `true` model cut.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This method changes the type of a previously defined constraint from ordinary constraint to model cut and vice versa.
2. Model cuts must be 'true' cuts, in the sense that they are redundant at the optimal MIP solution. The Optimizer does not guarantee to add all violated model cuts, so they must not be required to define the optimal MIP solution.
3. Constraint properties 'include vars', 'model cut', 'delayed constraint', and 'indicator constraint' are mutually exclusive. When changing from one of these types to another you must first reset the correponding type to 0.

Example The following example turns the constraint `Ctr3` into a model cut.

```
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);
```

Related topics Calls XPRBsetmodcut

setRange

Synopsis

```
int setRange(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 changes the type of a constraint to a range constraint within the bounds specified by `lw` and `up`. The constraint type and right hand side value of the constraint are replaced by the type `XPRB_R` (range) and the two bounds.

Example 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.

```
XPRBvar x,y;
XPRBctr Ctr1;
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 <= 400);
Ctr1.setRange(100,500);
Ctr1.addTerm(5);
if (Ctr1.getType() == XPRB_R)
{
 cout << "C1 in [" << Ctr1.getRangeL() << ",";
 cout << Ctr1.getRangeU() << "]" << endl;    // C1 in [105,505]
 cout << "C1 size: " << Ctr1.getSize() << endl;    // C1 size: 2
 Ctr1.setType(XPRB_G);
 Ctr1.print();                               // C1: 3*x + 2*y >= 505
}
```

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 `Ctr1` 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 Ctr1;
```

```
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 <= 400);
Ctr1.setTerm(5, y);        // C1:  3*x + 5*y <= 400
Ctr1.add(x+10);            // C1:  4*x + 5*y <= 390
Ctr1.setTerm(400);         // C1:  4*x + 5*y <= 400
Ctr1.addTerm(5);           // C1:  4*x + 5*y <= 405

cout << "Coefficient of x in " << Ctr1.getName() << ": ";
cout << Ctr1.getCoefficient(x) << endl;
cout << "Coefficient of x*y in " << Ctr1.getName() << ": ";
cout << Ctr1.getCoefficient(x,y) << endl;
```

Related topics Calls `XPRBsetterm` or `XPRBsetqterm`

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. Note that all terms in a cut must belong to the same problem as the cut itself.

Constructors

```
XPRBcut();
XPRBcut(xbcut *c);
XPRBcut(xbcut *c, XPRBrelation& r);
```

Methods

`int 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

```
XPRBcut();
XPRBcut(xbcut *c);
XPRBcut(xbcut *c, XPRBrelation& r);
```

Arguments

`c` A cut in BCL C.

`r` Linear relation defining the cut.

Description Create a new cut object.

Method detail

add

Synopsis

```
int add(XPRBexpr& le);
```

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 `XPRBcut.print`) are shown as comments. Please notice in particular the different behavior of `add` and `addTerm` 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

```
int addTerm(XPRBvar& var, double val);
int addTerm(double val, XPRBvar& var);
int addTerm(XPRBvar& var);
int addTerm(double val);
```

Arguments

`var` A BCL variable.

`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 cut, comprising the variable `var` with coefficient `val`. If the cut already has a term with variable `var`, `val` is added to its coefficient. If no variable is specified, the value `val` is added to the right hand side of the cut. Cut terms can also be added with method `XPRBcut.add`.

Example See `XPRBcut.add`.

Related topics Calls `XPRBaddcutterm`

delTerm

Synopsis

```
int delTerm(XPRBvar& var);
```

Argument

`var` 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 `XPRBcut.setTerm`.

Example See `XPRBcut.add`.

Related topics Calls `XPRBdelcutterm`

getCRef

Synopsis

```
xbcut *getCRef();
```

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

```
int getID();
```

Return value Classification or identification number.

Description This method returns the classification or identification number of a cut.

Example See `XPRBcut.setID`.

Related topics Calls `XPRBgetcutid`

getRHS

Synopsis

```
double getRHS();
```

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

```
int getType();
```

Return value

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

Description This method returns the type of the given cut.

Related topics Calls `XPRBgetcuttype`

isValid

Synopsis

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the cut object is correctly defined.

print

Synopsis

```
int print();
```

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

```
void reset();
```

Description Clear the definition of the cut object.

setID

Synopsis

```
int setID(int id);
```

Argument `id` 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 `XPRBprob.addCuts`.

Example This example defines a cut and then modifies its ID and relation type. The resulting output is shown in the comment.

```
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
```

Related topics Calls `XPRBsetcutid`

setTerm

Synopsis

```
int setTerm(XPRBvar& var, double val);
int setTerm(double val, XPRBvar& var);
int setTerm(XPRBvar& var);
int setTerm(double val);
```

Arguments

`var`	A BCL variable.
`val`	Value of the coefficient of the variable `var`.

Return value 0 if method executed successfully, 1 otherwise.

Description This function sets the coefficient of a variable to the value `val`. If no variable is specified, the right hand side of the cut is set to `val`.

Example See `XPRBcut.add`.

Related topics Calls `XPRBsetcutterm`

setType

Synopsis

```
int setType(int type);
```

Argument `type` Type of the cut:

`XPRB_L`	$\leq$ (inequality)
`XPRB_G`	$\geq$ (inequality)
`XPRB_E`	= (equation)

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 `XPRBprob.addCuts`.

Example See `XPRBcut.setID`.

Related topics Calls `XPRBsetcuttype`

XPRBexpr

Description

Methods and operators for constructing linear and quadratic expressions. Note that all variables in an expression must belong to the same problem.

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

```
int addTerm(XPRBvar& var, XPRBvar& var2, double val);
int addTerm(double val, XPRBvar& var, XPRBvar& var2);
int addTerm(XPRBvar& var, double val);
int addTerm(double val, XPRBvar& var);
int addTerm(XPRBvar& var);
int addTerm(double val);
```

Add a term 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`	BCL variables (may be the same).
`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`.

addTerm

Synopsis

```
int addTerm(XPRBvar& var, XPRBvar& var2, double val);
int addTerm(double val, XPRBvar& var, XPRBvar& var2);
int addTerm(XPRBvar& var, double val);
int addTerm(double val, XPRBvar& var);
int addTerm(XPRBvar& var);
int addTerm(double val);
```

Arguments

`var,var2` BCL decision variables (may be the same).

`val` A real value (coefficient).

Return value

The modified expression.

Description

This method adds a new term to an expression comprising the variable `var` (or the product of variables `var` and `var2`) with coefficient `val`. If the expression 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 constant term of the expression. Terms can also be added with method `XPRBexpr.add`.

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

```
int delTerm(XPRBvar& var);
int delTerm(XPRBvar& var, XPRBvar& var2);
```

Argument `var,var2` BCL decision variables (may be the same).

Return value The modified expression.

Description This function deletes a variable term from an expression. The constant term is changed or reset with method `XPRBexpr.setTerm`.

Example See `XPRBexpr.setTerm`.

getSol

Synopsis

```
double getSol();
```

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 `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 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 *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`.

mul

Synopsis

```
XPRBexpr& mul(double d);
XPRBexpr& mul(XPRBexpr& e);
```

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 `XPRBexpr.setTerm`.

neg

Synopsis

```
XPRBexpr& neg();
```

Return value Negation of the expression.

Description This method multiplies an expression with -1.

Example See `XPRBexpr.setTerm`.

setTerm

Synopsis

```
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);
```

Arguments

`var,var2`	BCL decision variables (may be the same).
`val`	A real value (coefficient).

Return value The modified expression.

Description This method sets the coefficient of a variable or of the product of the two specified variables to the value `val`. If no variable is specified, the constant term of the expression is set to `val`.

Example 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 `qe` after each modification.

```
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.addTerm(2, x);        //  - 1 + 5*x
qe -= 3*sqr(3*y);        //  - 1 + 5*x - 27*y^2
qe.neg();                //  1 - 5*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

```
XPRBindexSet();
XPRBindexSet(xbidxset *iset);
```

Argument `iset` An index set in BCL C.

Description Create a new index set object.

Method detail

addElement

Synopsis

```
int addElement(const char *text);
```

Argument `text` 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 ith element in the set if function executed successfully, `NULL` 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 `XPRBindexSet.addElement`.

Related topics Calls `XPRBgetidxelname`

getName

Synopsis

```
const char *getName();
```

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

Description This function returns the name of an index set.

Example See `XPRBindexSet.getSize`.

Related topics Calls `XPRBgetidxsetname`

getSize

Synopsis

```
int getSize();
```

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 This example displays the name, size, and complete contents of an index set.

```
XPRBprob prob("myprob");
XPRBindexSet ISet;

ISet = prob.newIndexSet("IS");
cout << ISet.getName() << " size: " << ISet.getSize() << endl;
ISet.print();
```

Related topics Calls `XPRBgetidxsetsize`

isValid

Synopsis

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the index set object is correctly defined. It should always be used to test the result returned by `XPRBprob.getIndexSetByName`.

Example See `XPRBprob.getIndexSetByName`.

print

Synopsis

```
int print();
```

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 `XPRBindexSet.getSize`.

Related topics Calls `XPRBprintidxset`

reset

Synopsis

```
void reset();
```

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.

`int addMIPSol(XPRBsol& sol, const char *name);`

`int addMIPSol(XPRBsol& sol);`
Add a new feasible, infeasible or partial MIP solution to the Optimizer.

`int beginCB(XPRSprob oprob);`
Start using the local optimizer problem with BCL in a callback.

`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 endCB();`
Reset BCL to the original optimizer problem in a callback.

`int exportProb(int format, const char *filename);`

`int exportProb(int format);`
Print problem matrix to a file.

`int fixGlobals(int ifround = 1);`
Fixes all the global entities to the values of the last found MIP solution.

`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.

`XPRSprob getXPRSprob();`
    Returns an XPRSprob 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 optimizer.

`int loadMIPSol(double *sol, int ncol, bool ifopt);`

`int loadMIPSol(double *sol, int ncol);`
    Load an integer solution into BCL or the Optimizer.

`int lpOptimize(const char *alg);`

`int lpOptimize();`
    Solve as a continuous problem.

`int mipOptimize(const char *alg);`

`int mipOptimize();`
    Solve as a discrete problem.

`bool nextCtr(XPRBctr& ref);`
    Enumerate constraints.

`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.

`XPRBsol newSol();`
    Create a solution.

`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 readBinSol(const char *filename = NULL, const char *flags = "");`
    >Read a solution from a binary solution file (.sol), loading it into the Optimizer.

`int readSlxSol(const char *filename = NULL, const char *flags = "");`
    >Read a solution from an ASCII solution file (.slx), loading it into the Optimizer.

`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 setName(const char *name);`
    Set the name of the problem.

`int setRealFmt(const char *fmt);`
    Set the format for printing real numbers.

`int setSense(int dir);`
    Set the sense of the optimization.

`int sync(int synctype);`
    Synchronize BCL with the Optimizer.

`int writeDir();`
`int writeDir(const char *filename);`
    Write directives to a file.

`int writeSol(const char *filename = NULL, const char *flags = "");`
    >Write the current Optimizer solution to a CSV format ASCII file, problem_name.asc (and .hdr).

`int writeBinSol(const char *filename = NULL, const char *flags = "");`
    >Write the current Optimizer solution to a binary solution file for later input into the Optimizer.

`int writePrtSol(const char *filename = NULL, const char *flags = "");`
    >Write the current Optimizer solution to a fixed format ASCII file, problem_name .prt.

`int writeSlxSol(const char *filename = NULL, const char *flags = "");`
    >Write the current Optimizer solution to an ASCII solution file (.slx) using a similar format to MPS files.

Constructor detail

XPRBprob

Synopsis

```
XPRBprob();
XPRBprob(const char *name);
```

Argument `name` The problem name. If none specified, BCL creates a unique name.

Description

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 `XPRBnewprob`

Method detail

addCuts

Synopsis

```
int addCuts(XPRBcut *cuts, int num);
```

Arguments

`cuts` Array of previously defined cuts.

`num` Number of cuts in `cuts`.

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 shows how to define the cut manager callback and add a cut to the Optimizer problem. The function call adding the cut is surrounded by the pair `XPRBprob.beginCB` and `XPRBprob.endCB` to coordinate BCL with the local Optimizer subproblem in the case of a multi-threaded MIP search.

```
int XPRS_CC cbNode(XPRSprob oprob, void *bp)
{
 XPRBprob *bprob = (XPRBprob*)bp;
 XPRBcut aCut;
 bprob->beginCB(oprob);
   ...                 // Define the cut 'aCut'
 bprob->addCuts(&aCut, 1);
 bprob->endCB();
 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.setSense(XPRB_MAXIM);
 prob.mipOptimize();
   ...                  // Solution output
}
```

Related topics Calls XPRBaddcuts

addMIPSol

Synopsis

```
int addMIPSol(XPRBsol& sol, const char *name);
int addMIPSol(XPRBsol& sol);
```

Arguments

`sol` A BCL solution.

`name` An optional name to associate with the solution. Can be NULL.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition, it is regenerated automatically before adding the solution.
2. The `XPRBprob.mipOptimize` method by default resets problem status including eventual loaded solutions; to avoid that, flag "c" should be specified for the `alg` argument of `XPRBprob.mipOptimize` when called after `XPRBprob.addMIPSol`.
3. The function returns immediately after passing the solution to the Optimizer. The solution is placed in a pool until the Optimizer is able to analyze the solution during a MIP solve.
4. If the provided solution is partial or found to be infeasible, a limited local search heuristic will be run in an attempt to find a close feasible integer solution. Values provided for continuous columns in partial solutions are currently ignored.
5. The *usersolnotify* callback can be used to discover the outcome of a loaded solution. The optional name provided as `name` will be returned in the callback.

Example Add a MIP solution for problem `prob` to the Optimizer.

```
XPRBprob prob("myprob");
XPRBsol sol = prob.newSol();
  ...         // Define + load the problem
prob.addMIPsol(sol, "myHeurSol");
/* Request notification of solution status after processing */
XPRSaddcbusersolnotify(prob.getXPRSprob(), solnotify_handler, &prob, 0);
```

Related topics Calls XPRBaddmipsol

beginCB

Synopsis

```
int beginCB(XPRSprob oprob);
```

Argument
`oprob` Reference to an Xpress Optimizer problem.

Return value
0 if method executed successfully, 1 otherwise.

Description
This function switches from the original problem to the specified (local) optimizer problem for all BCL accesses to Xpress Optimizer (in particular, for solution updates and the definition of cuts). A call to this function must precede any call to such BCL functions in optimizer MIP callbacks when the default multi-threaded MIP search is used for solving a problem. A call to `XPRBprob.beginCB` must always be matched by a call to `XPRBprob.endCB` to reset the optimizer problem within BCL and to release the BCL problem (access to the BCL problem is locked to the particular thread in between the two function calls).

Example
The example shows how to set up the integer solution callback of Xpress Optimizer to use BCL to display the results.

```
void XPRS_CC printSol(XPRSprob oprob, void *my_object)
{
 int num;
 XPRBprob *bprob = (XPRBprob*)bp;
 XPRBvar x;

 XPRSgetintattrib(oprob, XPRS_MIPSOLS, &num);
                        // Get number of the solution
 bprob->beginCB(oprob);  // Work with local optimizer problem
 bprob->sync(XPRB_XPRS_SOL);   // Update BCL solution values

 cout << "Solution " << num << ": Objective value: ";
 cout << bprob->getObjVal() << endl;
 x = bcl_prob->getVarByName("x_1");
 cout << x.getName() << ": " << x.getSol() << endl;

 bprob->endCB();         // Reset BCL to orig. optimizer problem
}

int main(int argc, char **argv)
{
 XPRBprob prob("myprob");
   ...                  // Define the problem
 XPRSsetcbintsol(prob.getXPRSprob(), printSol, &prob);
 prob.setSense(XPRB_MAXIM);
 prob.mipOptimize();
}
```

Related topics
Calls `XPRBbegincb`

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`. If the constraint occurs in a previously saved basis that is to be (re)loaded later on you should change its type to `XPRB_N` using `XPRBctr.setType` instead of entirely deleting the constraint.

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`

endCB

Synopsis

```
int endCB();
```

Return value 0 if method executed successfully, 1 otherwise.

Description This function switches back to the original optimizer problem for all BCL accesses to Xpress Optimizer. A call to this function terminates a block of calls to BCL functions in an optimizer callback that is preceded by `XPRBprob.beginCB`. The call to `XPRBprob.endCB` releases the BCL problem (access to the BCL problem is locked to the particular thread between the two function calls).

Example See `XPRBprob.beginCB`.

Related topics Calls `XPRBendcb`

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 `.mps`. 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.
3. The precision used by BCL for printing real numbers can be changed with `XPRB.setRealFmt` to obtain more accurate output for very large or very small numbers. For full precision matrix output the user is advised to switch to the Optimizer function `XPRSwriteprob`, preceded by a call to `XPRBprob.loadMat` (see Appendix B for further detail).

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

fixGlobals

Synopsis

```
int fixGlobals(int ifround = 1);
```

Argument `ifround` If all global entities should be rounded to the nearest discrete value in the solution before being fixed.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This is useful e.g. for finding the reduced costs for the continuous variables after the global variables have been fixed to their optimal values. The global variables are fixed to the value of the MIP solution only in the Optimizer (not in BCL).
2. In order to eventually resync the bounds of global variables to their original values defined in BCL (i.e. unfix them), a call to XPRBsync with the flag `XPRB_XPRS_PROB` can be used.

Example Performs a global search on problem expl2 and then uses XPRBfixglobals before solving the remaining linear problem:

```
XPRBprob prob("myprob");
  ...        // Define + load the problem
prob.mipOptmize();
prob.fixGlobals();
prob.lpOptimize();
```

Related topics Calls XPRBfixglobals

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

```
XPRBctr getCtrByName(const char *name);
```

Argument `name` 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 checked with `XPRBctr.isValid`. This method cannot be used if the names dictionary has been disabled (see `XPRBprob.setDictionarySize`).

Example The following retrieves a constraint by its name and if it has been found prints it out.

```
XPRBprob prob("myprob");
XPRBctr C2;

C2 = prob.getCtrByName("C2");
if (C2.isValid())  C2.print();
```

Related topics Calls `XPRBgetbyname`

getIndexSetByName

Synopsis

```
XPRBindexSet getIndexSetByName(const char *name);
```

Argument `name` 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 checked with `XPRBindexSet.isValid`. This method cannot be used if the names dictionary has been disabled (see `XPRBprob.setDictionarySize`).

Example The following retrieves an index by its name and if a set has been found prints it out.

```
XPRBprob prob("myprob");
XPRBindexSet I2;

I2 = prob.getIndexSetByName("IS");
if (I2.isValid())  I2.print();
```

Related topics Calls `XPRBgetbyname`

getLPStat

Synopsis

```
int getLPStat();
```

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`	LP problem is not solved.
`XPRB_LP_NONCONVEX`	QP problem is nonconvex.

Description The return value of this method provides LP status information from the Xpress Optimizer.

Example See `XPRBprob.mipOptimize`, `XPRBctr.getAct`.

Related topics Calls `XPRBgetlpstat`

getMIPStat

Synopsis

```
int getMIPStat();
```

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.
`XPRB_MIP_UNBOUNDED`	LP unbounded;

Description This methods returns the global (MIP) status information from the Xpress Optimizer.

Example See `XPRBprob.mipOptimize`.

Related topics Calls `XPRBgetmipstat`

getName

Synopsis

```
const char *getName();
```

Return value Name of the problem if function executed successfully, `NULL` 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 `XPRBgetprobname`

getNumIIS

Synopsis

```
int getNumIIS();
```

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.mipOptimize`.

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.lpOptimize, and XPRBprob.mipOptimize.

Related topics Calls XPRBgetsense

getSosByName

Synopsis

```
XPRBsos getSosByName(const char *name);
```

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 checked with XPRBsos.isValid. This method cannot be used if the names dictionary has been disabled (see XPRBprob.setDictionarySize).

Example The following retrieves a SOS by its name and if it has been found prints it out.

```
XPRBprob prob("myprob");
XPRBsos S2;

S2 = prob.getSosByName("SO2");
if (S2.isValid())  S2.print();
```

Related topics Calls XPRBgetbyname

getVarByName

Synopsis

```
XPRBvar getVarByName(const char *name);
```

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 checked with XPRBvar.isValid. This method cannot be used if the names dictionary has been disabled (see XPRBprob.setDictionarySize).

Example The following retrieves a variable by its name and if it has been found prints it out.

```
XPRBprob prob("myprob");
XPRBvar b2;

b2 = prob.getVarByName("b");
if (b2.isValid())
{ b2.print(); cout << endl; }
```

Related topics Calls XPRBgetbyname

getXPRSprob

Synopsis

```
XPRSprob getXPRSprob();
```

Return value Reference to a problem in Xpress Optimizer if executed successfully, `NULL` otherwise

Description This method returns an `XPRSprob` 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 since the basis has been stored (addition of variables and constraints is handled—deletion of constraints is not handled: instead of entirely deleting a constraint, change its type to `XPRB_N` using `XPRBctr.setType` if you wish to load the basis later on). 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`

loadMIPSol

Synopsis

```
int loadMIPSol(double *sol, int ncol, bool ifopt);
int loadMIPSol(double *sol, int ncol);
```

Arguments

`sol` Array of size `ncol` holding the solution values.

`ncol` Number of variables (continuous+discrete) in the problem.

`ifopt` Whether to load the solution into the Optimizer:
- `false` load into BCL only (default);
- `true` load solution into the Optimizer.

Return value

`0`	Solution accepted,
`1`	Solution rejected because it is infeasible,
`2`	Solution rejected because it is cut off,
`3`	Solution rejected because the LP reoptimization was interrupted,
`-1`	Solution rejected because an error occurred,
`-2`	The given solution array does not have the expected size,
`-3`	Error loading solution into BCL.

Description

1. This method loads a MIP solution from an external source (*e.g.*, the Xpress MIP Solution Pool) into BCL or the Optimizer. The solution is given in the form of an array, indexed by the column numbers of the decision variables. The size `ncol` of the array must correspond to the number of columns in the matrix (generated by a call to `XPRBprob.loadMat` or by starting an optimization run from BCL). If the solution is loaded into BCL the values are accepted as is, if the solution is loaded into the Optimizer (`ifopt = true`), the Optimizer will check whether the solution is acceptable and recalculates the values for the continuous variables in the solution. In the latter case the solution is loaded into BCL only once it has been successfully loaded and validated by the Optimizer.
2. If the matrix loaded in the Optimizer does not correspond to the current state of the specified problem definition, it is regenerated automatically before loading the solution.

Example

Load a MIP solution for problem `prob` into BCL, but not into the Optimizer. We know that the problem has 5 variables.

```
XPRBprob prob("myprob");
double vals[] = {1.5, 1, 0, 4, 2.2};
  ...         // Define + load the problem
if (prob.loadMIPSol(vals, 5)!=0)
 cout << "Loading the solution failed." << endl;
```

Related topics

Calls `XPRBloadmipsol`

lpOptimize

Synopsis

```
int lpOptimize(const char *alg);
int lpOptimize();
```

Argument

`alg` Choice of the solution algorithm and options, as a string of flags. If no flag is specified, solve the problem using the default LP/QP algorithm; otherwise, if the argument includes:

- `"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);
- `"c"` continue a previously interrupted optimization run.

Return value

0 if method executed successfully, 1 otherwise.

Description

This method selects and starts the Xpress Optimizer LP/QP solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, *e.g.* "`pn`. 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. The sense of the optimization (default: minimization) can be changed with function `XPRBprob.setSense`. Before solving a problem, the objective function must be selected with `XPRBprob.setObj`.

Example

See `XPRBprob.saveBasis`, `XPRBprob.mipOptimize`.

Related topics

Calls `XPRBlpoptimize`

mipOptimize

Synopsis

```
int mipOptimize(const char *alg);
int mipOptimize();
```

Argument

`alg` Choice of the solution algorithm and options, as a string of flags. If no flag is specified, solve the problem using the default MIP/MIQP algorithm; otherwise, if the argument includes:

- `"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 (for the initial LP);
- `"l"` stop after solving the initial continuous relaxation (using MIP information in presolve);
- `"c"` continue a previously interrupted optimization run.

Return value

0 if method executed successfully, 1 otherwise.

Description

This method selects and starts the Xpress Optimizer MIP/MIQP solution algorithm. The characters indicating the algorithm choice may be combined where it makes sense, *e.g.* "`pn`. 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. The sense of the optimization (default: minimization) can be changed with function `XPRBprob.setSense`. Before solving a problem, the objective function must be selected with `XPRBprob.setObj`.

Example

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.

```
XPRBprob prob("myprob");
  ...        // Define the problem
prob.setSense(XPRB_MAXIM);
prob.lpOptimize();
if (prob.getLPStat() == XPRB_LP_OPTIMAL)
 cout << "LP solution: " << prob.getObjVal() << endl;
prob.mipOptimize();
if (prob.getMIPStat() == XPRB_MIP_OPTIMAL ||
    prob.getMIPStat() == XPRB_MIP_SOLUTION)
 cout << "MIP solution: " << prob.getObjVal() << endl;
```

Related topics

Calls `XPRBmipoptimize`

nextCtr

Synopsis

```
bool nextCtr(XPRBctr& ref);
```

Argument

`ref` Reference constraint or `NULL`.

Return value

`true` if more constraints can be retrieved, `false` if the end of the enumeration has been reached.

Description This method is used to enumerate the constraints of a problem. The argument `ref` serves to keep track of the current location in the enumeration; if this parameter is `NULL`, the first constraint is returned, otherwise the constraint that follows the reference constraint is returned.

Example This code extract prints all constraints of the problem `prob`.

```
XPRBprob prob("myprob");
  ...         // Define + load the problem
XPRBctr iter = NULL;
while (prob.nextCtr(iter)) {
  iter.print();
}
```

Related topics Calls `XPRBgetnextctr`

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).

`id` 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 `XPRBprob.addCuts` from the Optimizer cutmanager callback.

Example The following example shows different possibilities of how to define cuts.

```
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);
```

Related topics Calls `XPRBnewcut`

newIndexSet

Synopsis

```
XPRBindexSet newIndexSet();
XPRBindexSet newIndexSet(const char *name);
XPRBindexSet newIndexSet(const char *name, int maxsize);
```

Arguments

`name` Name of the index set to be created. May be `NULL` if not required.

`maxsize` Maximum size of the index set.

Return value A new BCL index set.

Description This method 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 XPRBprob.setDictionarySize.)

Example The following example defines an index set of size 10 and then adds two elemnts to the set.

```
XPRBindexSet ISet;
XPRBprob prob("myprob");
int ind;

ISet = prob.newIndexSet("IS", 10);
ind = ISet.addElement("a"); ISet += "b";
```

Related topics Calls XPRBnewidxset

---

newSol

Synopsis

```
XPRBsol newSol();
```

Return value A new BCL solution.

Description This method creates a new solution.

Example See XPRBsol.setVar.

Related topics Calls XPRBnewsol

---

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 SO1, prints is out (output displayed as comment) and then deletes it. After this it defines an SOS-2 named SO2.

```
XPRBvar x,y,z;
XPRBsos SO1, SO2;
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);

SO1 = prob.newSos("SO1");
SO1.add(x+2*y+3*z);
SO1.print();             //  SO1(1): x(+1) y(+2) z(+3)
prob.delSos(SO1);

SO2 = prob.newSos("SO2", 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`

readBinSol

Synopsis

```
int readBinSol(const char *filename = NULL, const char *flags = "");
```

Arguments

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .bin will be appended.

`flags` Flags to control solution import. If no flags need to be specified, use either `NULL` or the empty string. Refer to function XPRSreadbinsol in the 'Xpress Optimizer Reference Manual' for details.

Return value 0 if method executed successfully, 1 otherwise.

Description This function reads a solution from a binary solution file (.sol), loading it into the Optimizer.

Example This example the reads a solution from file example2.sol

```
XPRBprob prob("example2");
  ...        // Define + load the problem
prob.readBinSol();
prob.mipOptimize();
```

Related topics Calls `XPRBreadbinsol`

readSlxSol

Synopsis

```
int readSlxSol(const char *filename = NULL, const char *flags = "");
```

Arguments

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .slx will be appended.

`flags` Flags to control solution import. If no flags need to be specified, use either `NULL` or the empty string. Refer to function XPRSreadslxsol in the 'Xpress Optimizer Reference Manual' for details.

Return value 0 if method executed successfully, 1 otherwise.

Description This function reads a solution from an ASCII solution file (.slx), loading it into the Optimizer.

Example This example the reads a solution from file example2.slx

```
XPRBprob prob("example2");
  ...        // Define + load the problem
prob.readSlxSol();
prob.mipOptimize();
```

Related topics Calls `XPRBreadslxsol`

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 spare matrix elements (`EXTRACOLS`, `EXTRAROWS`, and `EXTRAELEMS`) 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.lpOptimize();
```

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, preferrably a prime number; 0 disables the dictionary (for names dictionary only).

Return value 0 if method executed successfully, 1 otherwise.

Description

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 this dictionary is enabled (default setting) BCL automatically resizes this dictionary to a suitable size for your problem. If nevertheless you wish to set the size by yourself we recommend to choose a value close to the number of variables+constraints in your problem.
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 Calls `XPRBsetdictionarysize`

setMsgLevel

Synopsis

```
int setMsgLevel(int lev);
```

Argument

`level` The message level, *i.e.* 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

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 `XPRB`.

Example

The following example changes the global BCL message printing level to 'errors' only and sets the printing level for problem `prob` back to the default. It also modifies the values of the Optimizer printing controls for simplex and MIP logging.

```
XPRBprob prob("myprob");
XPRB::setMsgLevel(1);
prob.setMsgLevel(3);
XPRSsetintcontrol(prob.getXPRSprob(), XPRS_LPLOG, 0);
XPRSsetintcontrol(prob.getXPRSprob(), XPRS_MIPLOG, -500);
```

Related topics

Calls `XPRBsetmsglevel`

setObj

Synopsis

```
int setObj(XPRBctr ctr);
int setObj(XPRBexpr e);
int setObj(XPRBvar v);
```

Arguments

`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 `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.

Example

See `XPRBprob.newCtr`.

Related topics Calls XPRBsetobj

setName

Synopsis

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

Argument `name` A string of up to 1024 characters containing the new problem name.

Return value 0 if method executed successfully, 1 otherwise.

Description This method sets the problem name.

Related topics Calls XPRBsetprobname

setRealFmt

Synopsis

```
int setRealFmt(const char *fmt);
```

Argument `fmt` Format string (as used by the C function `printf`). Simple format strings are of the form `%n` where *n* may be, for instance, one of

`g`	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)
`.numf`	print real numbers in the style [-]d.d where the number of digits after the decimal point is equal to the given precision `num`.

Return value 0 if method executed successfully, 1 otherwise.

Description

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 XPRB.

Example See XPRB.setRealFmt.

Related topics Calls XPRBsetrealfmt

setSense

Synopsis

```
int setSense(int dir);
```

Argument `dir` Sense of the objective function, which must be one of:

`XPRB_MAXIM`	maximize the objective;
`XPRB_MINIM`	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 `XPRBprob.exportProb`.

Related topics Calls `XPRBsetsense`

sync

Synopsis

```
int sync(int synctype);
```

Argument `synctype` Type of the synchronization. Possible values:

`XPRB_XPRS_SOL`	update the BCL solution information with the LP solution currently held in the Optimizer;
`XPRB_XPRS_SOLMIP`	update the BCL solution information with the last MIP solution found by the Optimizer;
`XPRB_XPRS_PROB`	force problem reloading.

Return value 0 if method executed successfully, 1 otherwise.

Description

1. This method resets the BCL problem status.
2. `XPRB_XPRS_SOL`: retrieves the current LP solution (through `XPRSgetlpsol` function and `XPRS_LPOBJVAL` attribute); correctly used also in *intsol* callbacks as, when an integer solution is found during a global search, it is always set up as an LP solution to the current node.
3. `XPRB_XPRS_SOLMIP`: retrieves the last MIP solution found (through `XPRSgetmipsol` function and `XPRS_MIPOBJVAL` attribute); if used from an *intsol* callback, it will not necessarily return the solution that caused the invocation of the callback (it is possible that another thread finds a new integer solution before that one is retrieved).
4. `XPRB_XPRS_SOL` and `XPRB_XPRS_SOLMIP`: the solution information in BCL is updated with the solution held in the Optimizer at the next solution access (only the objective value is updated immediately).
5. `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.mipOptimize();
```

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`

writeSol

Synopsis

```
int writeSol(const char *filename = NULL, const char *flags = "");
```

Arguments

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If no file extension is specified, then extensions .hdr and .asc will be appended.

`flags` Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function XPRSwritesol in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if method executed successfully, 1 otherwise.

Description

This function writes out, to a CSV format ASCII file, the current Optimizer solution. If no file extension is specified, then two files will be written with extensions .asc and .hdr 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.

Example

This example the writes current solution to the file example2.asc (and .hdr).

```
XPRBprob prob("example2");
  ...         // Define + load the problem
prob.mipOptimize();
prob.writeSol();
```

Related topics

Calls `XPRBwritesol`

writeBinSol

Synopsis

```
int writeBinSol(const char *filename = NULL, const char *flags = "");
```

Arguments

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .sol will be appended.

`flags` Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function XPRSwritebinsol in the 'Xpress Optimizer Reference Manual' for details.

Return value 0 if method executed successfully, 1 otherwise.

Description This function writes out, to a binary file, the current Optimizer solution. If no file extension is specified, then the extension .sol 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.

Example This example the writes current solution to the file example2.sol.

```
XPRBprob prob("example2");
  ...         // Define + load the problem
prob.mipOptimize();
prob.writeBinSol();
```

Related topics Calls `XPRBwritebinsol`

writePrtSol

Synopsis

```
int writePrtSol(const char *filename = NULL, const char *flags = "");
```

Arguments

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .prt will be appended.

`flags` Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function XPRSwriteprtsol in the 'Xpress Optimizer Reference Manual' for details.

Return value 0 if method executed successfully, 1 otherwise.

Description This function writes out, to a fixed format ASCII file, the current Optimizer solution. If no file extension is specified, then the extension .prt 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.

Example This example the writes current solution to the file example2.prt.

```
XPRBprob prob("example2");
  ...         // Define + load the problem
prob.mipOptimize();
prob.writePrtSol();
```

Related topics Calls `XPRBwriteprtsol`

writeSlxSol

Synopsis

```
int writeSlxSol(const char *filename = NULL, const char *flags = "");
```

Arguments

`fname` Name of the solution file. May be `NULL` or the empty string if the problem name is to be used. If omitted, the extension .slx will be appended.

`flags` Flags to control output format. If no flags need to be specified, use either `NULL` or the empty string. Refer to function XPRSwriteslxsol in the 'Xpress Optimizer Reference Manual' for details.

Return value

0 if method executed successfully, 1 otherwise.

Description

This function writes out, to an ASCII solution file (using a similar format to MPS files) the current Optimizer solution. If no file extension is specified, then the extension .slx 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.

Example

This example the writes current solution to the file example2.slx.

```
XPRBprob prob("example2");
  ...         // Define + load the problem
prob.mipOptimize();
prob.writeSlxSol();
```

Related topics

Calls `XPRBwriteslxsol`

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.

XPRBsol

Description

Methods for defining, modifying and accessing solutions. Note that all variables in a solution must belong to the same problem as the solution itself.

Constructors

`XPRBsol();`

`XPRBsos(xbsol *s);`

Methods

`int delVar(XPRBvar& var);`
Delete a variable from a solution.

`int getSize();`
Get the size of a solution.

`int getVar(XPRBvar& var, double *val);`
Get the value assigned to a variable in a solution.

`bool isValid();`
Test the validity of the solution object.

`int print();`
Print out a solution

`void reset();`
Reset the solution object.

`int setVar(XPRBvar& var, double val);`
Set a variable to the given value in a solution.

Constructor detail

XPRBsol

Synopsis

```
XPRBsol();
XPRBsos(xbsol *s);
```

Argument

`s` A solution in BCL C.

Description

Create a new solution object.

Method detail

---

delVar

Synopsis

```
int delVar(XPRBvar& var);
```

Argument

`var` A BCL variable.

Return value 0 if method executed successfully, 1 otherwise.

Description This function deletes a variable (assigned to a value) from the given solution.

Example See `XPRBsol.setVar`.

Related topics Calls `XPRBdelsolvar`

getSize

Synopsis

```
int getSize();
```

Return value Size (= number of variables assigned to a value) of the solution, or -1 in case of an error.

Description This method returns the size of a solution (or -1 in case of an error).

Example See `XPRBsol.setVar`.

Related topics Calls `XPRBgetsolsize`

getVar

Synopsis

```
int getVar(XPRBvar& var, double *val);
```

Arguments

`var`	A BCL variable.
`val`	Pointer to a double where the value will be returned.

Return value

`0`	variable `var` is assigned a value in the solution and the value is returned in `val`;
`-1`	variable `var` is not assigned any value in the solution (`val` is left unmodified);
`1`	an error has occurred.

Description This method retrieves the value assigned to the given variable in a solution.

Example See `XPRBsol.setVar`.

Related topics Calls `XPRBgetsolvar`

isValid

Synopsis

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the solution object is correctly defined.

print

Synopsis

```
int print();
```

Return value 0 if function executed successfully, 1 otherwise.

Description This method prints out a solution (note that `XPRBsol` solutions represent user-defined solutions to be passed to the Optimizer, not solutions coming from the Optimizer). A solution is printed as a sequence like "*varname* = *value*, ... ". If the solution doesn't contain any variable, only an empty line is printed.. It is not available in the student version.

Example See `XPRBsol.setVar`.

Related topics Calls `XPRBprintsol`

reset

Synopsis

```
void reset();
```

Description This method clears the definition of the solution object; includes deletion of the underlying C object. This may be useful to free the memory used for storing the solution (note that this would be freed in any case when the problem is destroyed).

Example See `XPRBsol.setVar`.

Related topics Calls `XPRBdelsol`

setVar

Synopsis

```
int setVar(XPRBvar& var, double val);
```

Arguments

`var` A BCL variable.

`val` Value assigned to variable `var` in this solution.

Return value 0 if method executed successfully, 1 otherwise.

Description This function sets a variable to the given value in a solution. If the variable was already assigned a value, the value is overwritten.

Example This example sets variable x and y in solution `sol1` and then changes them.

```
XPRBvar x, y;
XPRBsol sol1;
XPRBprob prob("myprob");
double d;

x = prob.newVar("x", XPRB_PL, 0, 200);
```

```
y = prob.newVar("y", XPRB_PL, 0, 200);
sol1 = prob.newSol();
sol1.setVar(x, 7.0);
sol1.setVar(y, 15.0);
cout << "Sol1: ";
sol1.print();                        // Sol1: x = 7, y = 15
sol1.setVar(x, 10.0);
sol1.getVar(x, &d);
cout << "In sol1, x = " << d << endl; // In sol1, x = 10
sol1.delVar(y);
cout << "Sol1: ";
sol1.print();                        // Sol1: x = 10
sol1.reset();
```

Related topics Calls `XPRBsetsolvar`

XPRBsos

Description

Methods for modifying and accessing Special Ordered Sets and operators for constructing them. Note that all members in a SOS must belong to the same problem as the SOS itself.

Constructors

`XPRBsos();`

`XPRBsos(xbsos *s);`

`XPRBsos(xbsos *s, XPRBexpr& l);`

Methods

`int 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

```
int add(const XPRBexpr& le);
```

Argument `le` A linear expression.

Return value 0 if method executed successfully, 1 otherwise.

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 `XPRBsos.print`) and the output printed by the program are displayed as comments.

```
XPRBvar x,y,z;
XPRBsos SO1, SO2;
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);

SO1 = prob.newSos("SO1", XPRB_S1);
SO1.add(x+2*y+3*z);          // SO1(1): x(+1) y(+2) z(+3)
SO1 += 2*z-x;                // SO1(1): y(+2) z(+5)
cout << SO1.getName() << " type: ";
cout << (SO1.getType()==XPRB_S1?1:2) << endl;
                             // SO1 type: 1
SO2 = prob.newSos("SO2", XPRB_S2, 10*x+20*y);
SO2.addElement(z, 5);        // SO2(2): x(+10) y(+20) z(+5)
SO2.delElement(x);           // SO2(2): y(+20) z(+5)
```

addElement

Synopsis

```
int addElement(XPRBvar& var, double val);
int addElement(double val, XPRBvar& var);
```

Arguments
`var` Reference to a variable.
`val` 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 `XPRBsos.add`.

Related topics Calls `XPRBaddsosel`

delElement

Synopsis

```
int delElement(XPRBvar& var);
```

Argument `var` 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 `XPRBsos.add`.

Related topics Calls `XPRBdelsosel`

getCRef

Synopsis

```
xbsos *getCRef();
```

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

```
const char *getName();
```

Return value Name of the SOS if executed successfully, `NULL` 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 `XPRBsos.add`.

Related topics Calls `XPRBgetsosname`

getType

Synopsis

```
int getType();
```

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.

Description This method returns the type of a SOS.

Example See `XPRBsos.add`.

Related topics Calls `XPRBgetsostype`

isValid

Synopsis

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the SOS object is correctly defined. It should always be used to test the result returned by `XPRBprob.getSosByName`.

Example See `XPRBprob.getSosByName`.

print

Synopsis

```
int print();
```

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 `XPRBprob.getSosByName`.

Related topics Calls `XPRBprintsos`

setDir

Synopsis

```
int setDir(int type, double val);
int setDir(int type);
```

Arguments	`type`	The directive type, which must be one of: `XPRB_PR` priority; `XPRB_UP` first branch upwards; `XPRB_DN` first branch downwards; `XPRB_PU` pseudo cost on branching upwards; `XPRB_PD` pseudo cost on branching downwards.
	`val`	An argument dependent on the type of the directive being defined. If `type` is: `XPRB_PR` `val` will be the priority value, an integer between `1` (highest) and `1000` (lowest), the default; `XPRB_UP` no input is required; `XPRB_DN` no input is required; `XPRB_PU` `val` will be the value of the pseudo cost for the upward branch; `XPRB_PD` `val` will be the value of the pseudo cost for the downward branch.

Return value 0 if method executed successfully, 1 otherwise.

Description This method sets any type of branching directive available in Xpress. 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. Method `XPRBvar.setDir` may be used to set a directive for a variable.

Example See `XPRBprob.clearDir`.

Related topics Calls `XPRBsetsosdir`

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

```
XPRBvar();
XPRBvar(xbvar *v);
```

Argument `v` A variable in BCL C.

Description Create a new variable object.

Method detail

fix

Synopsis

```
int fix(double val);
```

Argument `val` 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 `XPRBvar.setLB` and `XPRBvar.setUB`. The value `val` 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 `XPRBfixvar`

getColNum

Synopsis

```
int getColNum();
```

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 `XPRBprob.getProbStat`. The counting of column numbers starts with `0`.

Example See `XPRBvar.getSol`.

Related topics Calls `XPRBgetcolnum`

getCRef

Synopsis

```
xbvar *getCRef();
```

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

```
double getLB();
```

Return value Lower bound on the variable (default 0).

Description This method returns the currently defined lower bound on a variable.

Example See `XPRBvar.getName`.

Related topics Calls `XPRBgetbounds`

getLim

Synopsis

```
double getLim();
```

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 `XPRBvar.getName`.

Related topics Calls `XPRBgetlim`

getName

Synopsis

```
const char *getName();
```

Return value Name of the variable if executed successfully, `NULL` 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 The following example displays information about a semi-continuous variable. The output printed by this program extract is shown in the comment.

```
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]
```

Related topics Calls XPRBgetvarname

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.
Reduced cost information is available only after LP solving. To obtain reduced cost values for a MIP solution (that is, if function XPRBprob.getMIPStat returns values XPRB_MIP_SOLUTION or XPRB_MIP_OPTIMAL), you need to fix the discrete variables to their solution values with a call to XPRSfixglobals, followed by a call to XPRBlpoptimize before calling XPRBgetrcost. Otherwise, if this function is called when a MIP solution is available it returns 0.
If no solution information is available this function outputs a warning and returns 0.
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. In this case it returns the reduced cost value in the last LP that has been solved.

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:

`XPRB_UPACT`	upper activity (= the activity level [solution value] that would result from a cost coefficient increase from the input cost to the upper cost XPRB_UCOST—assuming a minimization problem—ignoring the upper bound on this variable);
`XPRB_LOACT`	lower activity (= the activity level [solution value] that would result from a cost coefficient decrease from the input cost to the lower cost XPRB_LCOST—assuming a minimization problem—ignoring the upper bound on this variable);
`XPRB_UUP`	upper unit cost (= the change in the objective function per unit of change in the activity up to the upper activity XPRB_UPACT);
`XPRB_UDN`	lower unit cost (= the change in the objective function per unit of change in the activity down to the lower activity XPRB_LOACT)
`XPRB_UCOST`	upper cost;
`XPRB_LCOST`	lower cost.

Return value Ranging information of the required type.

Description

1. 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.
2. For non-basic variables, the unit costs are always the (absolute) values of the reduced costs.

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.lpOptimize();
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

XPRB_PL	continuous;
XPRB_BV	binary;
XPRB_UI	general integer;
XPRB_PI	partial integer;
XPRB_SC	semi-continuous;
XPRB_SI	semi-continuous integer;
-1	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

```
double getUB();
```

Return value Upper bound on the variable (default XPRB_INFINITY).

Description This method returns the currently defined upper bound on a variable.

Example See XPRBvar.getName.

Related topics Calls XPRBgetbounds

isValid

Synopsis

```
bool isValid();
```

Return value `true` if object is valid, `false` otherwise.

Description This method checks whether the variable object is correctly defined. It should always be used to test the result returned by `XPRBprob.getVarByName`.

Example See `XPRBprob.getVarByName`.

print

Synopsis

```
int print();
```

Return value The number of characters printed.

Description This method prints out a variable. It is not available in the student version.

Example See `XPRBprob.getVarByName`.

Related topics Calls `XPRBprintvar`

setDir

Synopsis

```
int setDir(int type, double val);
int setDir(int type);
```

Arguments

`type` Directive type, which must be one of:

`XPRB_PR`	priority;
`XPRB_UP`	first branch upwards;
`XPRB_DN`	first branch downwards;
`XPRB_PU`	pseudo cost on branching upwards;
`XPRB_PD`	pseudo cost on branching downwards.

`val` An argument dependent on the type of directive to be defined. Must be one of:

`XPRB_PR`	priority value — an integer between `1` (highest) and `1000` (least priority), the default;
`XPRB_UP`	no input required;
`XPRB_DN`	no input required;
`XPRB_PU`	value of the pseudo cost on branching upwards;
`XPRB_PD`	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. 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 possibly 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:

`XPRB_PL`	continuous;
`XPRB_BV`	binary;
`XPRB_UI`	general integer;
`XPRB_PI`	partial integer;
`XPRB_SC`	semi-continuous;
`XPRB_SI`	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 `eql` 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`, `XPRBsol`, `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 extend 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' located in subdirectory `docs/bcl/dhtml/javadoc` of the Xpress installation directory.

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`.
If a BCL Java model is embedded into an application we recommend to use explicit finalization on the `XPRBprob` object once it is no longer needed to free up the memory used by it (particularly, the memory used by the underlying C structures that are not taken into account by the automated garbage collection in Java). Alternatively, a problem can be `reset` to free up the memory used by the optimization and solution data without removing the problem definition itself.

```
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]).lEql(z) );

  p.newCtr("Prec", start[0].add(DUR[0]).lEql(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 */

  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.mipOptimize();               /* 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 */

  p.finalize();                   /* Delete the problem (optional) */
  p=null;
 }
}
```

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 `XPRBreadlinecb` and `XPRBreadarrlinecb` (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]).lEql(z) );

  p.newCtr("Prec", start[0].add(DUR[0]).lEql(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 */

  p.finalize();                    /* Delete the problem (optional) */
  p=null;
 }
}
```

6.1.2 QCQP Example

The following is an implementation with BCL Java of the QCQP example described in Section 3.5.1:

```
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()) .lEql(R[i]) );

/****SOLVING + OUTPUT****/
  prob.setSense(XPRB.MINIM);         /* Sense of optimization */
  prob.lpOptimize();                 /* 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());

  p.finalize();                      /* Delete the problem */
  p=null;
 }
}
```

6.1.3 Error handling

If an error occurs, BCL Java raises exceptions. A large majority of these execeptions 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 3.6. 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 after explicitly finalizing it to free up memory;
- finalization of BCL to release the license.

```
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)) .gEql(41));
  prob.newCtr("C2", x[0].add(x[1].mul(2)) .eql(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.lpOptimize();             /* 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 */
     prob.setOutputStream();        /* Re-establish standard output for prob */
     bcl.setOutputStream();         /* Re-establish standard output for BCL */
     f.close();

     prob.finalize();               /* Delete the problem */
     prob=null;

     bcl.finalize();                /* Release license */
     bcl=null;

     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 distribution (located in subdirectory `docs/bcl/dhtml/javadoc` of the Xpress installation directory).

`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.
`XPRBsol`	Methods for constructing, modifying and accessing solutions.

`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 `XPRBexpr`).
`XPRBerror`	Exception raised by BCL errors (extends `Error`).
`XPRBlicenseError`	Exception raised by BCL licensing errors (extends `XPRBerror`).
`XPRBlicense`	For OEM licensing.

All Java classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsol`, `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 `eql` 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 OEM licensing documentation.

CHAPTER 7

BCL in .NET

7.1 An overview of BCL in .NET

The .NET interface of BCL provides the full functionality of the C version, targeting .NET framework version 2.0 or higher. 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 .NET.

In .NET, the termwise definition of constraints is simplified by the overloading of the algebraic operators like '+', '−', '<=', or '==' as in the C++ interface. With these operators constraints may be written in a form that is close to an algebraic formulation. Also, for printing output, it is possible to use both .NET native IO functions or the `XPRBprob.printF` BCL function (which corresponds to the BCL C `XPRBprintf` function).

The names of classes and methods have been adapted to .NET naming standards: all .NET classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsol`, `XPRBsos`, `XPRBbasis`) take the same names, with the exception of `XPRBidxset` which becomes `XPRBindexSet`in .NET. The names of the methods are also changed by dropping the prefix `XPRB` and references to the type of the object, and each word is capitalized. For example, function `XPRBgetvarname` is turned into the method `getName` of class `XPRBvar`. Two exceptions are `XPRBreadlinecb` and `XPRBreadarrlinecb` which maintain these names as methods of the `XPRBprob` class and thus become `XPRBprob.XPRBreadline` and `XPRBprob.XPRBreadarrline`. The auto-completion feature in Visual Studio .NET can be used to obtain a full list of class methods and properties and prototypes of each method.

The BCL functionality is exposed through the `XPRB` and `XPRBprob` classes, which reside in the `BCL` namespace, which resides in the `xprbdn.dll` assembly.

Each source file that uses BCL should import the `BCL` namespace and should be compiled with a reference to `xprbdn.dll`. To add a reference to a Visual Studio .NET project, select **Add Reference** from the **Project** menu. Click the `Browse` button and locate `xprbdn.dll` in the `XpressMP/bin` folder.

Users of the .NET command line compilers, for example, `csc` for C#, or `vbc` for Visual Basic, can add a reference with the `/reference` option:

```
csc /reference:<path to xprbdn.dll> <your source file>
```

The C++ classes and their methods documented in section 5.2 correspond to a large extend to the classes defined by the .NET interface, with some additional classes in the .NET version. A comprehensive documentation of the BCL .NET interface is available as a separate 'HTML on-line documentation' located in subdirectory `docs/bcl/bcl.net/HTML` of the Xpress installation directory.

7.1.1 Example

An example of the use of BCL in .NET is the following, which again constructs the example described in Chapter 2. Contrary to the C and C++ versions, BCL .NET needs to be initialized explicitly by calling the static method `XPRB.init()`.
BCL models can take up large amounts of memory therefore, if a BCL .NET model is embedded into an application, we recommend to explicitly release the resources used by the `XPRBprob` object, once it is no longer needed, by calling `XPRB.Dispose()` (particularly, the memory used by the underlying C structures that are not taken into account by the automated garbage collection in .NET). Alternatively, a problem can be `reset` to free up the memory used by the optimization and solution data without removing the problem definition itself.

```
using BCL;

namespace Examples
{
  public class xbexpl1
  {
      const int NJ  =  4;              /* Number of jobs */
      const int NT  = 10;              /* Time limit */

      static double[] DUR = {3,4,2,2};   /* Durations of jobs   */

      static XPRBvar[] start = new XPRBvar[NJ];          /* Start times of jobs  */
      static XPRBvar[,] delta = new XPRBvar[NJ,NT];      /* Binaries for start times */
      static XPRBvar z;                  /* Maximum completion time (makespan) */

      static XPRBprob p;

      static void jobsModel()
      {
          XPRBexpr le;
          int j,t;

          /****VARIABLES****/
          /* Create start time variables */
          for(j=0;j<NJ;j++) start[j] = p.newVar("start");
          z = p.newVar("z",BCLconstant.XPRB_PL,0,NT); /* Declare the makespan variable */

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

          /****CONSTRAINTS****/
          for(j=0;j<NJ;j++)              /* Calculate maximal completion time */
              p.newCtr("Makespan", start[j]+DUR[j] <= z);

          p.newCtr("Prec", start[0]+DUR[0] <= start[2]);
          /* Precedence relation between jobs  */

          for(j=0;j<NJ;j++)              /* Linking start times and binaries  */
          {
              le = new XPRBexpr(0);
              for(t=0;t<(NT-DUR[j]+1);t++)  le += (t+1)*delta[j,t];
              p.newCtr("Link_" + (j+1), le == start[j]);
          }

          for(j=0;j<NJ;j++)              /* One unique start time for each job  */
          {
              le = new XPRBexpr(0);
              for(t=0;t<(NT-DUR[j]+1);t++)  le += delta[j,t];
              p.newCtr("One_" + (j+1), le == 1);
          }

          /****OBJECTIVE****/
          /* Define and set objective function */
```

```
            p.setObj(p.newCtr("OBJ", new XPRBrelation(z)));

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

        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(BCLconstant.XPRB_PR,10*(t+1));
                    /* Give highest priority to variables for earlier start times */

            p.setSense(BCLconstant.XPRB_MINIM);
            p.mipOptimize();                    /* Solve the problem as MIP */
            statmip = p.getMIPStat();       /* Get the MIP problem status */

            if((statmip == BCLconstant.XPRB_MIP_SOLUTION) ||
               (statmip == BCLconstant.XPRB_MIP_OPTIMAL))
            {  /* An integer solution has been found */
                Console.WriteLine("Objective: {0}", p.getObjVal());
                for(j=0;j<NJ;j++)
                {  /* Print the solution for all start times */
                  System.Console.WriteLine("{0}: {1}", start[j].getName(), start[j].getSol());
                  for(t=0;t<NT-DUR[j]+1;t++)
                    System.Console.Write("{0}: {1} ",delta[j,t].getName(),delta[j,t].getSol());
                  System.Console.WriteLine();
                }
            }
        }

        public static void Main()
        {
           XPRB.init();                      /* Initialize BCL  */
           p = new XPRBprob("Jobs");       /* Create a new problem */
           jobsModel();                      /* Basic problem definition */
           jobsSolve();                      /* Solve and print solution */
           return;
        }
    }
  }
```

The definition of SOS is similar to the definition of constraints.

```
static XPRBsos[] set = new XPRBsos[NJ]; /* Sets regrouping start times for jobs */
static XPRBprob p;

public static void jobsModel()
{

  ...
  for(j=0;j<NJ;j++)              /* Declare binaries for each job */
      for(t=0;t<(NT-DUR[j]+1);t++)
          delta[j,t] = p.newVar("delta" + (j+1) + (t+1),BCLconstant.XPRB_PL,0,1);

  /****SETS****/
  for(j=0;j<NJ;j++)
  {
      le = new XPRBexpr(0);
      for(t=0;t<(NT-DUR[j]+1);t++)  le += (t+1)*delta[j,t];
      set[j] = p.newSos("sosj",BCLconstant.XPRB_S1,le);
  }
}
```

Branching directives for the SOSs are added as follows.

```
for(j=0;j<NJ;j++)
    set[j].setDir(BCLconstant.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 definition */
p.exportProb(BCLconstant.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 `XPRBreadlinecb` and `XPRBreadarrlinecb` (that is, using '!' as commentary sign, and ',' as separators, and skip blanks and empty lines), we need to configure the data file access in .NET.

In the following program listing we leave out the method `jobsSolve` because it remains unchanged from the previous.

```
using System.IO;
using BCL;

namespace Examples
{
  public class xbexpl1i
  {
      const int MAXNJ = 4;             /* Max. number of jobs */
      const int NT  = 10;             /* Time limit */

      //Define XPRBDATAPATH to whatever folder you wish.
      const string XPRBDATAPATH = "../../data";
      const string DATAFILE = XPRBDATAPATH + "/jobs/durations.dat";

      /**** DATA ****/
      static int NJ = 0;                   /* Number of jobs read in */
      static double[] DUR = new double[MAXNJ]; /* 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;                   /* Maximum completion time (makespan) */

      XPRBprob p;                         /* BCL problem */

      static void readData()
      {
          string name;
          FileStream file;
          StreamReader fileStreamIn;

           /* Create a new index set */
          Jobs = p.newIndexSet("jobs", MAXNJ);


          file = new FileStream(DATAFILE, FileMode.Open, FileAccess.Read);
          fileStreamIn = new StreamReader(file);

          object[] tempobj = new object[2];
          while((NJ<MAXNJ) &&
           (p.XPRBreadarrline(fileStreamIn, 99, "{t} , {g} ", out tempobj, 1) == 2))
          {
              int dummy;
              name = (string)tempobj[0];
              DUR[NJ] = (double)tempobj[1];
              dummy = Jobs + name;
              NJ++;
```

```
            }

            fileStreamIn.Close();
            file.Close();

            System.Console.WriteLine("Number of jobs read: " + Jobs.getSize());
        }

        static void jobsModel()
        {
            XPRBexpr le;
            int j,t;

            /****VARIABLES****/
                /* Create start time variables (incl. bounds) */
            start = new XPRBvar[NJ];
            if(start==null)
            {
                System.Console.WriteLine("Not enough memory for 'start' variables.");
                return;
            }
            for (j = 0; j < NJ; j++)
               start[j] = p.newVar("start", BCLconstant.XPRB_PL, 0, NT - DUR[j] + 1);
            z = p.newVar("z",BCLconstant.XPRB_PL,0,NT);  /* Declare the makespan variable */

            delta = new XPRBvar[NJ, NT];
            for(j=0;j<NJ;j++)              /* Declare binaries for each job  */
                for(t=0;t<(NT-DUR[j]+1);t++)
                    delta[j,t] = p.newVar("delta"+Jobs.getIndexName(j)+"_"+(t+1),
                      BCLconstant.XPRB_BV);

            /****CONSTRAINTS****/
            for(j=0;j<NJ;j++)              /* Calculate maximal completion time */
                p.newCtr("Makespan", start[j]+DUR[j] <= z);

            p.newCtr("Prec", start[0]+DUR[0] <= start[2]);
                /* Precedence relation between jobs  */

            for(j=0;j<NJ;j++)              /* Linking start times and binaries  */
            {
                le = new XPRBexpr(0);
                for(t=0;t<(NT-DUR[j]+1);t++)  le += (t+1)*delta[j,t];
                p.newCtr("Link_" + (j+1), le == start[j]);
            }

            for(j=0;j<NJ;j++)              /* One unique start time for each job  */
            {
                le = new XPRBexpr(0);
                for(t=0;t<(NT-DUR[j]+1);t++)  le += delta[j,t];
                p.newCtr("One_" + (j+1), le == 1);
            }

            /****OBJECTIVE****/
            p.setObj(p.newCtr(z));                   /* Define and set objective function */

            jobsSolve();                   /* Solve the problem */
        }

        public static void Main()
        {
           XPRB.init();
           p = new XPRBprob("Jobs");      /* Create a new problem */
           readData();                    /* Read in the data */
           jobsModel();                   /* Basic problem definition */
        }
    }
}
```

7.1.2 QCQP Example

The following is an implementation with BCL .Net of the QCQP example described in Section 3.5.1:

```
using BCL;

namespace Examples
{
  public class xbexpl1i
  {
      const int MAXNJ = 4;             /* Max. number of jobs */
      const int NT  = 10;              /* Time limit */

      //Define XPRBDATAPATH to whatever folder you wish.
      const string XPRBDATAPATH = "../../data";
      const string DATAFILE = XPRBDATAPATH + "/jobs/durations.dat";

      /**** DATA ****/
      static int NJ = 0;                     /* Number of jobs read in */
      static double[] DUR = new double[MAXNJ]; /* 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;                    /* Maximum completion time (makespan) */

      XPRBprob p;                          /* BCL problem */

      static void readData()
      {
          string name;
          FileStream file;
          StreamReader fileStreamIn;

           /* Create a new index set */
          Jobs = p.newIndexSet("jobs", MAXNJ);


          file = new FileStream(DATAFILE, FileMode.Open, FileAccess.Read);
          fileStreamIn = new StreamReader(file);

          object[] tempobj = new object[2];
          while((NJ<MAXNJ) &&
           (p.XPRBreadarrline(fileStreamIn, 99, "{t} , {g} ", out tempobj, 1) == 2))
          {
              int dummy;
              name = (string)tempobj[0];
              DUR[NJ] = (double)tempobj[1];
              dummy = Jobs + name;
              NJ++;
          }

          fileStreamIn.Close();
          file.Close();

          System.Console.WriteLine("Number of jobs read: " + Jobs.getSize());
      }

      static void jobsModel()
      {
          XPRBexpr le;
          int j,t;

          /****VARIABLES****/
              /* Create start time variables (incl. bounds) */
          start = new XPRBvar[NJ];
          if(start==null)
          {
```

```
            System.Console.WriteLine("Not enough memory for 'start' variables.");
            return;
        }
        for (j = 0; j < NJ; j++)
           start[j] = p.newVar("start", BCLconstant.XPRB_PL, 0, NT - DUR[j] + 1);
        z = p.newVar("z",BCLconstant.XPRB_PL,0,NT);  /* Declare the makespan variable */

        delta = new XPRBvar[NJ, NT];
        for(j=0;j<NJ;j++)              /* Declare binaries for each job  */
            for(t=0;t<(NT-DUR[j]+1);t++)
                delta[j,t] = p.newVar("delta" + Jobs.getIndexName(j) + "_" + (t+1),
                  BCLconstant.XPRB_BV);

        /****CONSTRAINTS****/
        for(j=0;j<NJ;j++)              /* Calculate maximal completion time */
            p.newCtr("Makespan", start[j]+DUR[j] <= z);

        p.newCtr("Prec", start[0]+DUR[0] <= start[2]);
            /* Precedence relation between jobs  */

        for(j=0;j<NJ;j++)              /* Linking start times and binaries  */
        {
            le = new XPRBexpr(0);
            for(t=0;t<(NT-DUR[j]+1);t++)  le += (t+1)*delta[j,t];
            p.newCtr("Link_" + (j+1), le == start[j]);
        }

        for(j=0;j<NJ;j++)              /* One unique start time for each job  */
        {
            le = new XPRBexpr(0);
            for(t=0;t<(NT-DUR[j]+1);t++)  le += delta[j,t];
            p.newCtr("One_" + (j+1), le == 1);
        }

        /****OBJECTIVE****/
        p.setObj(p.newCtr(z));                   /* Define and set objective function */

        jobsSolve();                   /* Solve the problem */
    }

    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(BCLconstant.XPRB_PR,10*(t+1));
                /* Give highest priority to variables for earlier start times */

        p.setSense(BCLconstant.XPRB_MINIM);
        p.mipOptimize();                   /* Solve the problem as MIP */
        statmip = p.getMIPStat();       /* Get the MIP problem status */

        if((statmip == BCLconstant.XPRB_MIP_SOLUTION) ||
           (statmip == BCLconstant.XPRB_MIP_OPTIMAL))
        {  /* An integer solution has been found */
          System.Console.WriteLine("Objective: " + p.getObjVal());
          for(j=0;j<NJ;j++)
          {                              /* Print the solution for all start times */
            System.Console.WriteLine(start[j].getName() + ": " + start[j].getSol());
            for(t=0;t<NT-DUR[j]+1;t++)
              System.Console.Write(delta[j,t].getName()+ ": "+delta[j,t].getSol()+" ");
            System.Console.WriteLine();
          }
        }
    }

    public static void Main()
    {
```

```
            XPRB.init();
            p = new XPRBprob("Jobs");      /* Create a new problem */
            readData();                    /* Read in the data */
            jobsModel();                   /* Basic problem definition */
        }
    }
}
```

7.1.3 Error handling

If an error occurs, BCL .NET behaves like the C interface, that is, it prints an error message and terminates the program. Alternatively, if BCL error handling is disabled by calling `XPRBprob.setErrCtrl(0)`, then all error messages are sent to the user-defined error callback without terminating the program; the user can both check these error messages and the return codes of each method to verify if it has completed correctly. The only case where a `BCLExceptions` is raised is when an error occurs while constructing a `BCLexpr` object. Below we show a .NET implementation of the example of user error handling with BCL from Section 3.6. 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;

```
using System;
using BCL;

namespace Examples
{
  public class UGExpl3
  {
      public static int rtsbefore = 1;

      public void modexpl3(ref XPRBprob prob)
      {
          XPRBvar[] x = new XPRBvar[3];
          XPRBctr[] ctr = new XPRBctr[2];
          XPRBexpr cobj;
          int i;

          for(i=0;i<2;i++)
              x[i] = prob.newVar("x_"+i, BCLconstant.XPRB_UI, 0, 100);

          /* Create the constraints:
          C1: 2x0 + 3x1 >= 41
          C2:  x0 + 2x1  = 13 */
          XPRBexpr C1linexp = new XPRBexpr();

          XPRBexpr C2linexp = new XPRBexpr();
          C1linexp = 2 * x[0] + 3 * x[1];
          C2linexp = x[0] + 2 * x[1];
          prob.newCtr("C1", C1linexp >= 41);
          prob.newCtr("C2", C2linexp == 13);

          /* Uncomment the following line to cause an error in the model that
          triggers the user error handling: */

          // x[3] = prob.newVar("x_2", BCLconstant.XPRB_UI, 10, 1);

          /* Objective: minimize x0+x1 */
          cobj = new XPRBexpr(0);
          for(i=0;i<2;i++) cobj += x[i];
          prob.setObj(prob.newCtr("OBJ", cobj));
          prob.setSense(BCLconstant.XPRB_MINIM); /* Set objective sense to minimization */
          prob.print();                          /* Print current problem definition */
```

```
    prob.lpOptimize();                      /* Solve the LP */
    prob.printF("Problem status: " + prob.getProbStat() +
        "  LP status: " + prob.getLPStat() + "  MIP status: " +
        prob.getMIPStat() + "\n");

    /* This problem is infeasible, that means the following command will fail.
     * It prints a warning if the message level is at least 2 */

    prob.printF("Objective: " + prob.getObjVal() + "\n");

    /* Print solution values */
    for(i=0;i<2;i++)
        prob.printF(x[i].getName() + ":" + x[i].getSol() + ", ");
    prob.printF("\n");
}

/***********************************************************************/

/**** User error handling function ****/
public static void usererror(IntPtr prob, object vp, int num, int type, string t)
{
    Exception eBCL = new Exception("Error in usererror().");
    System.Console.WriteLine("BCL error " +num+ ": " + t);
    if(type==BCLconstant.XPRB_ERR) throw eBCL;
}

/**** User printing function ****/
public static void userprint(IntPtr prob, object vp, string msg)
{
    /* Print 'BCL output' whenever a new output line starts,
    otherwise continue to print the current line. */
    if(rtsbefore==1) System.Console.Write("BCL output: " + msg);
    else System.Console.Write(msg);
    rtsbefore = (msg.Length>0 && msg[msg.Length-1]=='\n') ? 1 : 0;
}

/***********************************************************************/

// This is where one might add custom logging
static void DoSomeErrorLogging(string msg)
{
    Console.WriteLine("Here's an error message! {0}", msg);
}

public static int Main()
{
    try
    {
        /* Switch to error handling by the user's program */
        XPRB.setErrCtrl(0); // no auto quit on error
        int initCode = XPRB.init();
        if (initCode != 0 && initCode != 32) // both values are valid
        {
            DoSomeErrorLogging(Optimizer.XPRS.GetLicErrMsg());
            return initCode;
        }
        UGExpl3 TestInstance = new UGExpl3();

        XPRBprob prob = new XPRBprob("EXPL3");
        if (!prob.isValid())
        {
            DoSomeErrorLogging("Unable to create XPRBprob \"EXPL3\"");
            return 1;
        }

        /* Set the printing flag. Try other values:
                0 - no printed output, 1 - only errors,
                2 - errors and warnings, 3 - all messages */
        prob.setMsgLevel(2);
```

```
                    /* Define the printing callback function */
                    prob.MessageCallbacks += new XPRBMessageCallback(userprint);

                    try
                    {
                        prob.ErrorCallbacks += new XPRBErrorCallback(usererror);
                        TestInstance.modexpl3(ref prob); /* Formulate and solve the problem */
                        System.Console.WriteLine("I'm about to exit cleanly");
                        return 0;
                    }
                    catch
                    {
                        System.Console.WriteLine("I cannot build the problem");
                        return 1;
                    }

                }
                catch
                {
                    System.Console.WriteLine("I cannot create the problem");
                    return 1;
                }
            }
        }
    }
```

7.2 .NET class reference

The complete set of classes of the BCL .NET interface is summarized in the following list. For a detailed documentation of the .NET interface the reader is referred to the BCL .NET HTML NDoc that is part of the Xpress distribution (located in subdirectory `docs/bcl/bcl.net/HTML` of the Xpress installation directory).

`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.
`XPRBsol`	Methods for constructing, modifying and accessing solutions.
`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 `XPRBexpr`).
`XPRBterm`	Methods for initialisation and handling of XPRBterm objects.
`XPRBVersion`	Version number in its encoded form.

`BCLconstant`	All BCL related constants.
`BCLExceptions`	Methods for BCL Exceptions.
`Scanner`	Class for reading file data in to objects in a similar manner to scanf().

All .NET classes that have a direct correspondence with modeling objects in BCL (namely `XPRBprob`, `XPRBvar`, `XPRBctr`, `XPRBcut`, `XPRBsol`, `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 .NET problem by using method `getXPRSprob` of class `XPRBprob`, please see Section B.8 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 the same result codes.

Two important classes that do 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 class `BCLconstant` that contains the definition of all constant parameters. This means, any parameter with the prefix `XPRB_` in standard BCL is referred to as a constant member of the .NET class `BCLconstant`. For example, `XPRB_BV` in standard BCL becomes `BCLconstant.XPRB_BV` in .NET.

In .NET, 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 couple of other additional classes are related to error handling and data input, namely `BCLExceptions` which represent exceptions raised when errors occur while building constraints; and `Scanner` which is used internally to implement the `XPRBprob.XPRBreadline` and `XPRBprob.XPRBreadarrline` methods.

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-1501 ***(fct) No active problem.***
Function *fct* has been called with an invalid problem argument. Check whether the problem has been created (function `XPRBnewprob`).

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 basisi 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 `XPRBlpoptimize` or `XPRBmipoptimize` include `'b'`, `'d'`, `'g'`, `'l'`, `'n'`, `'p'`, `'c'`. Refer to the reference manual for details.

E-1524 ***(fct) Xpress Optimizer error* num *during* 'name'. *Return value:* val.**
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* and return value *val*.

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.

W-1527 ***Loading MIP solution failed. Return value:* val.**
The specified MIP solution has not been loaded into BCL. Please see the documentation of function `XPRBloadmipsol` for an explanation of the return values.

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-1538 *(***fct***) Setting* **'descr'** *can only be applied to standard constraints.*
'Model cut', 'delayed constraint', 'include vars' and 'indicator constraint' are mutually exclusive flags. A constraint for which one of these flags is set cannot be turned into one of the other types without previously resetting the corresponding flag (using the appropriate function `XPRBsetincvars`, `XPRBsetmodcut`, `XPRBsetdelayed`, or `XPRBsetindicator` with argument value 0).

E-1539 *(***fct***) Incorrect constraint type for indicator constraint* **'name'***.*
Indicator constraints must be inequalities or range constraints.

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 ***XPRS:* text**
Refer to the Optimizer Reference Manual for the indicated error.

E-1571 **text**
The initialization has not found a valid license.

E-1572 ***(*fct*) No Xpress Optimizer problem given.***
The function *fct* requires an argument of type `XPRSprob`.

W-1573 ***'*fct1*' without matching '*fct2*'.***
The indicated functions (*e.g.*, `XPRBbegincb` / `XPRBendcb`) must always be used as a pair.

E-1575 ***(*fct*) Unexpected argument value* val.**
The value *val* lies outside the accepted range of values for the indicated function.

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.

W-1592 ***(*fct*) Setting limit for variable* ‘name’ *of wrong type.***
Cannot set the integer limit because the given variable is not a partial integer, semi-continuous or semi-continuous integer variable.

W-1593 ***(*fct*) Setting limit* lim *for variable* ‘name’ *outside bounds (*bdl,bdu*).***
The new integer limit is outside the bounds for the given variable or is negative.

E-1594 ***(*fct*) Setting* ‘descr’ *can only be applied to non-binding (type N) constraints.***
Only a non-binding (type `XPRB_N`) constraint can be set to be an ’Include vars’ special constraint.

E-1595 ***(*fct*) Incorrect constraint type for include vars constraint* ‘name’.**
Include vars constraints must be non-binding constraints (constraints of type `XPRB_N`).

W-1596 ***Adding MIP solution failed. Error loading matrix (*val*).***
The specified MIP solution has not been added because the current problem definition could not be loaded into the Optimizer.

E-1597 ***(*fct*) No solution given.***
Function *fct* requires a solution of type `XPRBsol` as an input parameter. Check whether the solution has been created (function `XPRBnewsol`).

W-1598 ***(*fct*) No solution given.***
Function *fct* requires a solution of type `XPRBsol` as an input parameter. Check whether the solution has been created (function `XPRBnewsol`).

W-1599 ***(*fct*) Removing var* name*, set to a NaN value, from solution.***
Variables cannot be assigned `NaN` values (not a number) in a solution. Therefore variable *name* is removed from the solution (if present) to avoid ambiguities with its previously assigned value.

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 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—not for parallelized MIP—it needs to be updated by calling `XPRBsync` with the parameter `XPRB_XPRS_SOL` or `XPRB_XPRS_SOLMIP`):

- *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*: `XPRSreaddirs`, `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...`, `XPRSdel...`;
- *solution algorithms*: `XPRSlpoptimize`, `XPRSmipoptimize`;
- *input of data or problem(s)*: `XPRSreadprob`, `XPRSloadlp`, `XPRSloadglobal`, `XPRSloadqglobal`, `XPRSloadqp`, `XPRSalter`, `XPRSsetprobname`;
- *manipulation of the matrix*: `XPRSrestore`;
- callback: `XPRSsetcbmessage`

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 preferrable 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 `XPRBgetXPRSprob`, 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) */
XPRBsetsense(prob, XPRB_MAXIM);  /* Set sense to maximization */
XPRBlpoptimize(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 */
XPRSchgobjsense(opt_prob,XPRS_OBJ_MAXIMIZE);  /* Select maximization */
XPRSlpoptimize(opt_prob,"");     /* Maximize the LP problem */
XPRSgetintattrib(opt_prob, XPRS_ORIGINALCOLS, &cols);
                                 /* Get the number of columns */
sol = malloc(cols * sizeof(double));
XPRSgetlpsol(opt_prob, 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 XPRBgetcolnum correspond to the position of variables and constraints in the unpresolved 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.

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 ADDCUTTOFF value */
XPRBsetsense(bcl_prob, XPRB_MAXIM);  /* Select maximization */
XPRBlpoptimize(bcl_prob,"");       /* Load matrix and solve as 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 */
XPRBsetsense(bcl_prob, XPRB_MAXIM);  /* Select maximization */
```

```
XPRBlpoptimize(bcl_prob,"");   /* Load matrix and solve as 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;
  XPRBbegincb(bcl_prob, opt_prob);
                              /* Use local Optimizer problem in BCL */
  XPRSgetintattrib(opt_prob, XPRS_MIPSOLS, &num);
                              /* Get number of the solution */
  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));

  XPRBendcb(bcl_prob);        /* Reset BCL to main problem */
}

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 */
  XPRBsetsense(bcl_prob, XPRB_MAXIM);  /* Select maximization */
  XPRBmipoptimize(bcl_prob,"");   /* Solve as MIP problem */
}
```

The synchronization between BCL and the Optimizer during the MIP search requires some special care. The code extract above shows how to use the functions `XPRBbegincb` and `XPRBendcb` to coordinate the BCL solution information with the Optimizer subproblem in a default multi-threaded MIP search. Alternatively, you may choose to disable parallelism by setting the `XPRS_MIPTHREADS` control to 1.

```
 opt_prob = XPRBgetXPRSprob(bcl_prob);
 XPRSsetintcontrol(opt_prob, XPRS_MIPTHREADS, 1);
                              /* Use single-threaded MIP */
```

MIP solution information can also be accessed through the Optimizer library functions whereby it is possible to use the column or row indices saved for BCL modeling objects as shown below. In this case there is no need for synchronizing the BCL solution information.

```
void XPRS_CC printsol(XPRSprob opt_prob,void *my_object)
{
 int num,ncol;
 XPRBprob bprob;
 XPRBvar x;
 double *sol, objval;
```

```
 bprob = (XPRBprob)vp;
 XPRSgetintattrib(oprob, XPRS_ORIGINALCOLS, &ncol);
                                          /* Get the number of columns */
 sol = (double *)malloc(ncol * sizeof(double));
                                     /* Create the solution array */
 XPRSgetintattrib(oprob, XPRS_MIPSOLS, &num); /* Get number of the solution */
 XPRSgetlpsol(oprob, sol, NULL, NULL, NULL);  /* Get the solution values */
 XPRSgetdblattrib(oprob, XPRS_LPOBJVAL, &objval);
 printf("Solution %d: Objective value: %g\n", num, objval);
 x = XPRBgetbyname(bprob, "x_1", XPRB_VAR);
 if(XPRBgetcolnum(x)>-1)
   printf("%s: %g\n", XPRBgetvarname(x), sol[XPRBgetcolnum(x)]);
 free(sol);
}
```

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 ADDCUTTOFF value
XPRSgetintattrib(opt_prob,XPRS_ORIGINALROWS, &rows);
                               // Get number of rows
bcl_prob.setSense(XPRB_MAXIM); // Select maximization
bcl_prob.lpOptimize();         // Maximize the LP problem
```

Using Xpress Optimizer callbacks (multi-threaded MIP):

```
void XPRS_CC printsol(XPRSprob opt_prob,void *my_object)
{
  XPRBprob *bcl_prob
  XPRBvar x;
  int num;

  bcl_prob = (XPRBprob*)my_object;
  bcl_prob->beginCB(opt_prob); // Use local Optimizer problem in BCL
  XPRSgetintattrib(opt_prob, XPRS_MIPSOLS, &num);
                               // Get number of the solution
  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;
  bcl_prob->endCB();           // Reset BCL to main problem
}

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 = bcl_prob.newVar("x_1", XPRB_BV); // Define a variable
     ...                     // Define the rest of the problem
  opt_prob = bcl_prob.getXPRSprob();
  XPRSsetcbintsol(opt_prob, printsol, &bcl_prob);
                             // Define an integer solution callback
  bcl_prob.setSense(XPRB_MAXIM);       // Select maximization
  bcl_prob.mipOptimize();      // Maximize the MIP problem
}
```

The code extract below shows how to access MIP solution information directly through the Optimizer library functions using the column or row indices saved for BCL modeling objects. In this case there is no need for synchronization of BCL with the local solution information.

```
void XPRS_CC printsol(XPRSprob opt_prob,void *my_object)
{
  XPRBprob *bcl_prob
  XPRBvar x;
  int num, ncol;
  double *sol, objval;

  bcl_prob = (XPRBprob*)my_object;
  XPRSgetintattrib(opt_prob, XPRS_ORIGINALCOLS, &ncol);
                             // Get the number of columns
  sol = new double[ncol];      // Create the solution array
  XPRSgetintattrib(opt_prob, XPRS_MIPSOLS, &num);
                             // Get number of the solution
  XPRSgetlpsol(opt_prob, sol, NULL, NULL, NULL);  // Get the solution
  XPRSgetdblattrib(opt_prob, XPRS_LPOBJVAL, &objval);
  cout << "Solution " << num << ": Objective value: " << objval << endl;
  x = bcl_prob->getVarByName("x_1");
  if(x.getColNum()>-1)         // Test whether variable is in the
                             // matrix
    cout << x.getName() << ": " << sol[x.getColNum()] << endl;
  delete [] sol;
}
```

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 `XPRSprobException` and `XPRSexception`):

```
int rows;
XPRB bcl;
XPRSprob 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.getXPRSprob();
opt_prob.setIntControl(XPRS.MAXTIME, 60);
                                /* Set a time limit of 60 seconds */
opt_prob.setDblControl(XPRS.MIPADDCUTOFF, 0.999);
                                /* Set an ADDCUTTOFF value */
rows = opt_prob.getIntAttrib(XPRS.ORIGINALROWS);
                                /* Get number of rows */
bcl_prob.setSense(XPRB.MAXIM); // Select maximization
bcl_prob.lpOptimize();         // Maximize the LP problem
```

Using Xpress Optimizer callbacks (multi-threaded MIP):

```
static class IntSolCallback implements XPRSintSolListener
{
  public void XPRSintSolEvent(XPRSprob opt_prob, Object my_object)
  {
    XPRBprob bcl_prob
    XPRBvar x;
    int num;

    bcl_prob = (XPRBprob)my_object;
    try {
      bcl_prob.beginCB(opt_prob);
                                /* Use local Optimizer problem in BCL */
      num = opt_prob.getIntAttrib(XPRS.MIPSOLS);
                                /* Get number of the solution */
      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());
      bcl_prob.endCB();         /* Reset BCL to main problem */
    }
    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.setSense(XPRB.MAXIM);      /* Select maximization */
  bcl_prob.mipOptimize();             /* Maximize the MIP problem */
}
```

The code extract below shows how to access MIP solution information directly through the Optimizer library functions using the column or row indices saved for BCL modeling objects. In this case there is no need for synchronization of BCL with the local solution information.

```
static class IntSolCallback implements XPRSintSolListener
{
  public void XPRSintSolEvent(XPRSprob opt_prob, Object my_object)
  {
    XPRBprob bcl_prob
    XPRBvar x;
    int num;
    double [] sol;

    bcl_prob = (XPRBprob)my_object;
    try {
      ncol = opt_prob.getIntAttrib(XPRS.ORIGINALCOLS);
                              /* Get the number of columns */
      sol = new double[ncol];
      opt_prob.getSol(sol, null, null, null);   /* Get the solution */
      num = opt_prob.getIntAttrib(XPRS.MIPSOLS);
                              /* Get number of the solution */
      System.out.println("Solution " + num + ": Objective value: " +
                        opt_prob.getDblAttrib(XPRS.LPOBJVAL));
      x = bcl_prob.getVarByName("x_1");
      if(x.getColNum()>-1)     /* Test whether variable is in the
                                 matrix */
        System.out.println(x.getName() + ": " + sol[x.getColNum()]);
      sol = null;
    }
    catch(XPRSprobException e) {
      System.out.println("Error " + e.getCode() + ": " + e.getMessage());
    }
  }
}
```

B.8 Using the Optimizer with BCL .NET

Using the Optimizer with BCL .NET is very similar to the other interfaces already seen and the same considerations regarding BCL-compatible functions remain true. The BCL .NET interface also requires an explicit intialization of the Optimizer .NET interface (by calling `XPRS.Init()`) before the Optimizer problem is accessed.

The following are .NET implementations of the code extracts showing the use of BCL-compatible functions:

Setting and accessing parameters:

```
int rows;
XPRB.init();                     // Initialize BCL
XPRS.Init();                     // Initialize Xpress Optimizer
XPRSprob opt_prob;
XPRBprob bcl_prob;
bcl_prob = new XPRBprob("Example1"); // Create a new problem in BCL
   ...                           // Define the problem
bcl_prob.loadMat();
opt_prob = bcl_prob.getXPRSprob();
opt_prob.MaxTime = 60;           // Set a time limit of 60 seconds
opt_prob.MIPAddCutoff = 0.999; // Set an ADDCUTTOFF value
rows = opt_prob.OriginalRows;  // Get number of rows
bcl_prob.setSense(BCLconstant.XPRB_MAXIM); // Select maximization
bcl_prob.lpOptimize();           // Maximize the LP problem
```

Using Xpress Optimizer callbacks (multi-threaded MIP):

```
public class IntSolExample
  public static void PrintSolution(XPRSprob opt_prob, object my_object)
  {
    XPRBprob bcl_prob = (XPRBprob)my_object;
    bcl_prob.beginCB(opt_prob);
    int num = opt_prob.MIPSols;
    bcl_prob.sync(BCLconstant.XPRB_XPRS_SOL); // Update BCL solution values
    System.Console.WriteLine("Solution "+num+": Objective value: "+bcl_prob.getObjVal());
    XPRBvar x = bcl_prob.getVarByName("x_1");
    if( x.getColNum() > -1 ) // Test whether variable is in the matrix
      System.Console.WriteLine(x.getName() + ": " + x.getSol());
    bcl_prob.endCB();
  }

  public static void Main()
  {
    XPRB.init();
    XPRS.Init();
    XPRBprob bcl_prob = new XPRBprob("Example1");
    XPRBvar x = bcl_prob.newVar("x_1", BCLconstant.XPRB_BV); // Define a variable
       ...                                      // Define the rest of the problem
    XPRSprob opt_prob = bcl_prob.getXPRSprob();
    // Define an integer solution callback
    IntsolCallback printsol = new IntsolCallback(PrintSolution);
    opt_prob.AddIntsolCallback(printsol, (object)bcl_prob);
    bcl_prob.setSense(BCLconstant.XPRB_MAXIM);
    bcl_prob.mipOptimize();                     // Maximize the MIP problem
  }
}
```

The code extract below shows how to access MIP solution information directly through the Optimizer library functions using the column or row indices saved for BCL modeling objects. In this case there is no need for synchronization of BCL with the local solution information.

```
public class IntSolExample2
  public static void XPRSIntSolEvent(XPRSprob opt_prob, object my_object)
  {
    XPRBprob bcl_prob = (XPRBprob)my_object;
    int ncol = opt_prob.OriginalCols; // Get the number of columns
    double[] sol = new double[ncol];
    opt_prob.GetLpSol(sol);           // Get the solution
    int num = opt_prob.MIPSols;       // Get number of the solution
    System.Console.WriteLine("Solution {0}: Objective value: {1}",num,opt_prob.LPObjVal);
    XPRBvar x = bcl_prob.getVarByName("x_1");
    if( x.getColNum() >= 0 ) // Test whether variable is in the matrix
      System.Console.WriteLine(x.getName() + ": " + sol[x.getColNum()]);
  }
}
```

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 Reference 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:

`XPRBaddcutarrterm`	Add multiple linear terms to a cut.	p. 34
`XPRBaddcuts`	Add cuts to a problem.	p. 35
`XPRBaddcutterm`	Add a term to a cut.	p. 36
`XPRBdelcut`	Delete a cut definition.	p. 53
`XPRBdelcutterm`	Delete a term from a cut.	p. 54
`XPRBgetcutid`	Get the classification or identification number of a cut.	p. 79
`XPRBgetcutrhs`	Get the RHS value of a cut.	p. 80
`XPRBgetcuttype`	Get the type of a cut.	p. 81
`XPRBnewcut`	Create a new cut.	p. 134
`XPRBnewcutarrsum`	Create a sum cut with individual coefficients (i ci xi).	p. 135
`XPRBnewcutprec`	Create a precedence cut (v1+dur v2).	p. 136
`XPRBnewcutsum`	Create a sum cut (i xi).	p. 137

XPRBprintcut	Print out a cut.	p. 150
XPRBsetcutid	Set the classification or identification number of a cut.	p. 168
XPRBsetcutmode	Set the cut mode.	p. 169
XPRBsetcutterm	Set a cut term.	p. 170
XPRBsetcuttype	Set the type of a cut.	p. 171

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 */
 XPRBbegincb(bprob, oprob);              /* Coordinate BCL and Optimizer */
 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");
 }
 XPRBendcb(bprob);                       /* Reset BCL to main problem */
 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 */
```

```
 XPRBmipoptimize(prob, "");               /* Solve the MIP problem */
 ...                                      /* Solution output */
 return 0;
}
```

When using the default multi-threaded MIP search it is important to coordinate BCL with the local Optimizer subproblem by surrounding the calls to BCL functions in the cut manager callback by calls to `XPRBbegincb` and `XPRBendcb`. Alternatively, you may choose to disable parallelism by setting the `XPRS_MIPTHREADS` control to 1.

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
 bprob->beginCB(oprob);                // Coordinate BCL and Optimizer
 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;
 }
 bprob->endCB();                       // Reset BCL to main problem
 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.mipOptimize();                      // 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
   {
    bprob.beginCB(oprob);              /* Coordinate BCL and Optimizer */
    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) .lEql(start[0]), 2);
     ca[1] = bprob.newCut(start[2].mul(4) .add(start[3].mul(-5.3)) .lEql(-17), 2);
     System.out.println("Adding constraints:");
     for(i=0;i<2;i++) ca[i].print();
     bprob.addCuts(ca);
    }
    bprob.endCB();                     /* Reset BCL to main problem */
   }
   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.mipOptimize();                     /* Solve the problem as MIP */
```

```
  ...                                /* Solution output */
  }
 }
```

C.4 .NET version of the example

As is explained in Section B.8, before accessing directly the problem held in Xpress Optimizer we need to initialize explicitly the Optimizer .NET. The cut manager callback is implemented in this .NET example by the method `usrcme`.

```
using Optimizer;
using BCL;

namespace Examples
{
  public class xbcutex
  {
      static XPRBvar[] start = new XPRBvar[4]; // Start times of jobs

      public static int usrcme(XPRSprob xprsp, object vd)
      {
          XPRBcut[] ca = new XPRBcut[2];
          XPRBprob xprbp = (XPRBprob)vd;       // Get the BCL problem
          xprbp.beginCB(xprsp);                // Coordinate BCL and Optimizer

          int num = xprsp.Nodes;
          if(num == 2)                         // Only generate cuts at node 2
          {
              ca[0] = xprbp.newCut(start[1]+2 <= start[0], 2);
              ca[1] = xprbp.newCut((4*start[2]) - (5.3*start[3]) <= -17, 2);
              System.Console.WriteLine("Adding constraints:");
              for(int i=0;i<2;i++) ca[i].print();
              if(xprbp.addCuts(ca,2) != 0)
                System.Console.WriteLine("Problem with adding cuts.");
          }
          xprbp.endCB();                       // Reset BCL to main problem
          return 0;                            // Call this function once per node
      }

      public static void Main()
      {
         XPRB.init();                          // Initialize BCL
         XPRS.Init();                          // Initialize Optimizer

         XPRBprob xprbp = new XPRBprob("Jobs");// Create a new problem
         for(int j=0;j<4;j++)                  // Create 'start' variables
           start[j] = xprbp.newVar("start");
         ...   // Define constraints and an objective function
         XPRSprob xprsp = p.getXPRSprob();     // Get Optimizer problem
         xprbp.setCutMode(1);                  // Enable the cut mode
         CutmgrCallback del = new CutmgrCallback(usrcme);
         xprsp.AddCutmgrCallback(del, (object)xprbp); // Define the cut manager callback
         xprbp.mipOptimize();                  // Solve the problem as MIP
      }
  }
}
```

APPENDIX D

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information on the Product Support home page (www.fico.com/support).

On the Product Support home page, you can also register for credentials to log on to FICO Online Support, our web-based support tool to access Product Support 24x7 from anywhere in the world. Using FICO Online Support, you can enter cases online, track them through resolution, find articles in the FICO Knowledge Base, and query known issues.

Please include *'Xpress'* in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Sales and maintenance

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

About FICO

FICO (NYSE:FICO) delivers superior predictive analytics solutions that drive smarter decisions. The company's groundbreaking use of mathematics to predict consumer behavior has transformed entire industries and revolutionized the way risk is managed and products are marketed. FICO's innovative solutions include the FICO® Score—the standard measure of consumer credit risk in the United States—along with industry-leading solutions for managing credit accounts, identifying and minimizing the impact of fraud, and customizing consumer offers with pinpoint accuracy. Most of the world's top banks, as well as leading insurers, retailers, pharmaceutical companies, and government agencies, rely on FICO solutions to accelerate growth, control risk, boost profits, and meet regulatory and competitive demands. FICO also helps millions of individuals manage their personal credit health through www.myfico.com. Learn more at www.fico.com. FICO: Make every decision count™.

Index