How to Contact Dash

USA, Canada and all Americas
Dash Optimization Inc

Information and Sales: info@dashoptimization.com
Licensing: license-usa@dashoptimization.com
Product Support: support-usa@dashoptimization.com
Tel: +1 (201) 567 9445
Fax: +1 (201) 567 9443

Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Japan
Dash Optimization Japan

Information and Sales: info@jp.dashoptimization.com
Licensing: license@jp.dashoptimization.com
Product Support: support@jp.dashoptimization.com
Tel: +81 43 297 8836
Fax: +81 43 297 8827

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

Worldwide
Dash Optimization Ltd

Information and Sales: info@dashoptimization.com
Licensing: license@dashoptimization.com
Product Support: support@dashoptimization.com
Tel: +44 1926 315862
Fax: +44 1926 315854

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

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

1 Introduction
   1.1 An overview of Xpress-BCL .................................................. 1
   1.2 Note for Optimizer library users .......................................... 1
   1.3 Structure of this manual ..................................................... 2
   1.4 Conventions used ............................................................. 2

I Modeling with BCL

2 Modeling with BCL
   2.1 Problem handling ............................................................. 4
      2.1.1 Initialization and termination ........................................ 4
      2.1.2 Problem creation and deletion ....................................... 5
      2.1.3 Other basic functions ................................................. 5
      2.1.4 Input and output settings .......................................... 5
      2.1.5 Error handling ......................................................... 5
   2.2 Variables ............................................................................. 6
      2.2.1 Basic functions .......................................................... 6
      2.2.2 Variable arrays ......................................................... 6
   2.3 Constraints .......................................................................... 7
      2.3.1 Basic functions .......................................................... 7
      2.3.2 Predefined constraint functions .................................... 8
      2.3.3 Objective function .................................................... 8
   2.4 Solving and solution information .......................................... 9
   2.5 Example ............................................................................. 10
      2.5.1 Model formulation using basic functions ......................... 10
      2.5.2 Using variable arrays ................................................. 11
      2.5.3 Completing the example: problem solving and output .......... 12

3 Further modeling topics
   3.1 Data input and index sets .................................................... 14
      3.1.1 Example .................................................................. 14
   3.2 Special Ordered Sets ........................................................... 16
      3.2.1 Basic functions ........................................................ 16
      3.2.2 Array-based SOS definition ....................................... 17
      3.2.3 Example ................................................................. 17
   3.3 Output and printing .............................................................. 18
      3.3.1 Example .................................................................. 18
   3.4 Quadratic Programming with BCL ........................................ 19
      3.4.1 Example .................................................................. 20
   3.5 User error handling .............................................................. 20

II BCL library and class reference

4 BCL C library functions
   4.1 Layout for function descriptions ......................................... 24
      XPRBaddarrterm ................................................................. 25
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPRBaddcuts</td>
<td>26</td>
</tr>
<tr>
<td>XPRBaddcutterm</td>
<td>27</td>
</tr>
<tr>
<td>XPRBaddcutarrterm</td>
<td>28</td>
</tr>
<tr>
<td>XPRBaddidxl</td>
<td>29</td>
</tr>
<tr>
<td>XPRBaddqterm</td>
<td>30</td>
</tr>
<tr>
<td>XPRBadddsosel</td>
<td>31</td>
</tr>
<tr>
<td>XPRBaddsosear</td>
<td>32</td>
</tr>
<tr>
<td>XPRBaddterm</td>
<td>33</td>
</tr>
<tr>
<td>XPRBapparrvarel</td>
<td>34</td>
</tr>
<tr>
<td>XPRBcleardir</td>
<td>35</td>
</tr>
<tr>
<td>XPRBdefcbdelvar</td>
<td>36</td>
</tr>
<tr>
<td>XPRBdefcberr</td>
<td>37</td>
</tr>
<tr>
<td>XPRBdefcbmsg</td>
<td>38</td>
</tr>
<tr>
<td>XPRBdelarrvar</td>
<td>39</td>
</tr>
<tr>
<td>XPRBdelbasis</td>
<td>40</td>
</tr>
<tr>
<td>XPRBdelctr</td>
<td>41</td>
</tr>
<tr>
<td>XPRBdelcut</td>
<td>42</td>
</tr>
<tr>
<td>XPRBdelcutterm</td>
<td>43</td>
</tr>
<tr>
<td>XPRBdelprob</td>
<td>44</td>
</tr>
<tr>
<td>XPRBdelqobj</td>
<td>45</td>
</tr>
<tr>
<td>XPRBdelsos</td>
<td>46</td>
</tr>
<tr>
<td>XPRBdelsosel</td>
<td>47</td>
</tr>
<tr>
<td>XPRBdeltterm</td>
<td>48</td>
</tr>
<tr>
<td>XPRBendarrvar</td>
<td>49</td>
</tr>
<tr>
<td>XPRBexportprob</td>
<td>50</td>
</tr>
<tr>
<td>XPRBfinish, XPRBfree</td>
<td>51</td>
</tr>
<tr>
<td>XPRBfixvar</td>
<td>52</td>
</tr>
<tr>
<td>XPRBgetact</td>
<td>53</td>
</tr>
<tr>
<td>XPRBgetarrvarname</td>
<td>54</td>
</tr>
<tr>
<td>XPRBgetarrvarsize</td>
<td>55</td>
</tr>
<tr>
<td>XPRBgetbounds</td>
<td>56</td>
</tr>
<tr>
<td>XPRBgetbyname</td>
<td>57</td>
</tr>
<tr>
<td>XPRBgetcolnum</td>
<td>58</td>
</tr>
<tr>
<td>XPRBgetctrname</td>
<td>59</td>
</tr>
<tr>
<td>XPRBgetctrrng</td>
<td>60</td>
</tr>
<tr>
<td>XPRBgetctrtype</td>
<td>61</td>
</tr>
<tr>
<td>XPRBgetcutid</td>
<td>62</td>
</tr>
<tr>
<td>XPRBgetcutrh</td>
<td>63</td>
</tr>
<tr>
<td>XPRBgetcuttype</td>
<td>64</td>
</tr>
<tr>
<td>XPRBgetdual</td>
<td>65</td>
</tr>
<tr>
<td>XPRBgetidxel</td>
<td>66</td>
</tr>
<tr>
<td>XPRBgetidxelname</td>
<td>67</td>
</tr>
<tr>
<td>XPRBgetidxsetname</td>
<td>68</td>
</tr>
<tr>
<td>XPRBgetidxsetsizesize</td>
<td>69</td>
</tr>
<tr>
<td>XPRBgetiis</td>
<td>70</td>
</tr>
<tr>
<td>XPRBgetlim</td>
<td>71</td>
</tr>
<tr>
<td>XPRBgetlpstat</td>
<td>72</td>
</tr>
<tr>
<td>XPRBgetmipstat</td>
<td>73</td>
</tr>
<tr>
<td>XPRBgetmodcut</td>
<td>74</td>
</tr>
<tr>
<td>XPRBgetnumiis</td>
<td>75</td>
</tr>
<tr>
<td>XPRBgetobjyal</td>
<td>76</td>
</tr>
<tr>
<td>XPRBgetprobname</td>
<td>77</td>
</tr>
<tr>
<td>XPRBgetprobsstat</td>
<td>78</td>
</tr>
<tr>
<td>XPRBgetrange</td>
<td>79</td>
</tr>
<tr>
<td>XPRBgetrcost</td>
<td>80</td>
</tr>
<tr>
<td>XPRBgetrhs</td>
<td>81</td>
</tr>
<tr>
<td>XPRBgetrownum</td>
<td>82</td>
</tr>
<tr>
<td>XPRBgetsense</td>
<td>83</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

1.1 An overview of Xpress-BCL

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

Model formulation using Xpress-BCL is constraint-oriented. Such constraints may be built up either coefficient-wise, incrementally adding terms to a linear combination 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 XPRBlodmat explicitly transforms the constraint-wise representation in BCL into the matrix representation required by the Optimizer library. It is usually not necessary to call
this function because BCL automatically carries out this transformation whenever required.

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

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

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

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

1.3 Structure of this manual

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

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

1.4 Conventions used

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

- square brackets [...] contain optional material;
- curly brackets {...} contain optional material, of which one must be chosen;
- entities in italics which appear in expressions stand for meta-variables. The description following the meta-variable describes how it is to be used;
- the vertical bar symbol | is found on many keyboards as a vertical line with a small gap in the middle, but often confusingly displays on screen without the small gap in the middle. In UNIX it is referred to as the pipe symbol. Note that this symbol is not the same as the character sometimes used to draw boxes on a PC screen. In ASCII, the | symbol is 7C in hexadecimal, 124 decimal.
I. Modeling with BCL
Chapter 2
Modeling with BCL

2.1 Problem handling

2.1.1 Initialization and termination

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

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

2.1.2 Problem creation and deletion

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

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

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

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

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

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

```c
XPRBprob pb1;
...
pb1 = XPRBnewprob("Problem1");
```

Delete problem definition

```c
XPRBdelprob(pb1);
```

Delete solution information

```c
XPRBresetprob(pb1);
```

Load problem matrix

```c
XPRBloadmat(pb1);
```

Fix column ordering

```c
XPRBsetcolorder(pb1,1);
```

Get problem name

```c
XPRBgetprobname(pb1);
```

Figure 2.1: Creating, accessing and deleting problems in BCL

### 2.1.3 Other basic functions

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

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

### 2.1.4 Input and output settings

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

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

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

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

### 2.1.5 Error handling

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

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

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

2.2 Variables

2.2.1 Basic functions

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

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

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

2.2.2 Variable arrays

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

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

```c
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

```c
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

```c
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

```c
XPRBgetarrvarname(av2);
XPRBgetarrvarsize(av1);
```

Delete a variable array

```c
XPRBdelarrvar(av2);
```

Find by name

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

Branching directives

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

Setting bounds

```c
XPRBsetlb(y,4);
XPRBsetub(s[0],9);
XPRBfixvar(av2[2],6);
XPRBsetlim(y,5);
```

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

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

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

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

## 2.3 Constraints

### 2.3.1 Basic functions

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

Since all functions for constraint definition identify the corresponding constraint via its model
\[ \sum_{i=0}^{3} s_i \leq 20 \]

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

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

\text{XPRBnewarrsum(prob,"S2",s,D, XPRB_E,9);} 
\text{XPRBaddarrterm(ctr,s,D);} 
\text{XPRBaddterm(ctr,NULL,9);} 

\[ s_0 + D_0 \leq y \]

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

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

name, constraint definitions may be nested.

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

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

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

### 2.3.2 Predefined constraint functions

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

### 2.3.3 Objective function

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

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.
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);

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 bd1, bdu;
XPRBgetctrname(c);
XPRBgetrange(c,&bd1,&bdu);
XPRBgetrownum(c);
XPRBgetctrtype(c);
XPRBsetctrtype(c,XPRB_L);
XPRBgetmcut(c);
XPRBsetmcut(c,1);

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

Solve active problem
XPRBsolve(prob,"dg");
XPRBminim(prob,"pl");
XPRBmaxim(prob,"t");

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

Get objective value
XPRBgetobjval(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);

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

Figure 2.6: Solving and solution information

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

2.4 Solving and solution information

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

Before any solution function is called, the objective function must be selected using XPRBsetobj. The function XPRBsolve also requires the sense of the objective to be set, that is, whether to minimize (default) or to maximize the objective. All solution functions XPRBsolve, XPRBminim, and XPRBmaxim can be parameterized to choose the type of solution algorithm.

Once the problem has been solved, the following solution information can be obtained: the optimal objective function value (XPRBgetobjval), values for all the problem variables (XPRB-
getsol), slack values (XPRBgetslack), reduced costs (XPRBgetrcost), constraint activity (XPRBgetact), and dual values (XPRBgetdual). It is also possible to obtain ranging information for variables (XPRBgetvarrng) and constraints (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 (see Appendix B for more detail).

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

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

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

2.5 Example

The following example is an extract of a scheduling problem: four jobs with different durations need to 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 DURj is given. We define decision variables startj representing the start time of jobs and binary variables deltajt indicating whether job j starts in time period t (deltajt = 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

```c
#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_PB, "delta", 0, 1);

    for(j=0;j<NJ;j++) /* Define precedence relationship */
        for(t=0;t<NT;t++)
            if (t < NT-DUR[j])
                delta[j][t] = delta[j][t+DUR[j]];

    for(j=0;j<NJ;j++) /* Define constraint for each job */
        for(t=0;t<NT;t++)
            if (t >= start[j])
                delta[j][t] = 1;

    /* Define constraint to make the 'dummy job' start at t=NT */
    delta[NJ][NT-1] = 1;

    prob->XPRBsync(XPRB_XPRS_SOL); /* Update BCL solution information */
    XPRBgetoptprob(prob, &optprob);
    XPRBgetprobstat(prob, &stat);
    printf("Optimum value: %f\n", optprob->value);
    printf("Status: %d\n", stat);
}
```

Modeling with BCL 10 BCL Reference Manual
\[ \text{delta}[j][t] = \text{XPRBnewvar}\left(\text{prob}, \text{XPRB}\_BV, \text{"delta"}, 0, 1\right); \]

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 \(\text{start} \) and \(\text{delta}_j\) by arrays of variables \(\text{start} \) and \(\text{delta}_j\). 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.

```c
#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 */
```
The set of constraints \textit{Link} (linking start time variables and binaries) can also be formulated using arrays and the constraint relation \texttt{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 \texttt{av[NJ]}.

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

\begin{verbatim}
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. */
    XPRBnewarrsum(prob, "Link", v, ind, XPRB_E, 0); /* Def. constraint using array v */
    XPRBdelarrvar(v); /* Free the allocated memory */
}
\end{verbatim}

\section*{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, \texttt{jobs_solve}. To complete the program we write a main that calls the model definition and the solution functions.

\begin{verbatim}
void jobs_solve(void)
{
    int statmip;
    int j;

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

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

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

   for(j=0;j<NJ;j++)
      for(t=0;t<NT-DUR[j]+1;t++)
         XPRBsetvardir(delta[j][t], XPRB_PR, 10*(t+1));
   /* Give highest priority to var.s for earlier start times */
Chapter 3
Further modeling topics

3.1 Data input and index sets

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

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

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

3.1.1 Example

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

```
FILE *datafile;
char name[50];
double dval, dvals[5];
XPRBreadarrlinecb(XPRB_FGETS,datafile,200,"T,d",name,&dval);
XPRBreadarrlinecb(XPRB_FGETS,datafile,200,"d;",dvals,5);

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

XPRBaddidxel(set1,"Prob1");

int size, ind;
ind = XPRBgetidxel(set1,"Prod1");
nname = XPRBgetidxelname(set1,14);
nname = 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.
names, together with the durations, are given in a separate data file, durations.dat:

- ABC14, 3
- DE45F, 4
- GH9I99, 2
- KLMN789, 2

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

- ABC14, 3 ! product1, duration
- DE45F, & ! this line is continued
- 4 ! in the next line
- GH9I99, 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.

```c
#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); }
```
# 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 XPRBdellosel and XPRBdelsos respectively. BCL also has functions to retrieve the name of a SOS and its type (XPRBgetsosname and XPRBgetsostype). It is also possible to set branching directives for a SOS (function XPRBsetsosdir), including priorities, choice of the preferred branching direction and definition of pseudo costs.

3.2.2 Array-based SOS definition

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

3.2.3 Example

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

```c
XPRBprob prob;  /* BCL problem */
XPRBvar delta[NJ][NT];  /* Variables for start times */
XPRBkos 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).

```c
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.

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

3.3 Output and printing

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

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

3.3.1 Example

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

```c
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.

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

#define NJ 4 /* Number of jobs */
#define NT 10 /* Time limit */
do double DUR[] = {3,4,2,2}; /* Durations of jobs */
XPRBvar start[NJ]; /* Start times of jobs */
XPRBprob prob; /* BCL problem */
...

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

### 3.4 Quadratic Programming with BCL

As an extension to LP and MIP, BCL also provides support for formulating and solving Quadratic Programming (QP) 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.

In BCL, the quadratic part of the objective function is defined termwise, much like constraint definition. The coefficient of a quadratic term is either set to a given value (\texttt{XPRBsetqterm}) or its value is augmented by the given value (\texttt{XPRBaddqterm}). Since the quadratic part of the objective function is treated separately from the linear part, it has to be deleted with a call to \texttt{XPRBdelqobj} if the objective is deleted or changed to a different constraint. Note that the definition of the quadratic objective terms should always be preceded by the definition of the corresponding variables.

Unless BCL is used in Student Mode, functions \texttt{XPRBprintprob}, \texttt{XPRBexportprob}, and \texttt{XPRBprintctr} (if applied to the objective function) will print or output the complete problem definition to a file, including the quadratic part of the objective.

### 3.4.1 Example

The following code fragment defines the quadratic objective function

\[-6x_0 + 2x_0^2 - 2x_0x_1 + x_1^2\]

It is assumed that the variables $x_i$ have been created earlier in the program.

```c
XPRBctr cobj;
XPRBvar x[2];
...
cobj = XPRBnewctr(prob,"OBJ",XPRB_N); /* Create the objective
   constraint */
XPRBaddterm(prob,cobj, x[0],-6); /* Add the linear term */
XPRBsetobj(prob,cobj); /* Choose the obj. function */
XPRBaddqterm(prob,x[0],x[0],2); /* Add quadratic terms */
XPRBaddqterm(prob,x[0],x[1],-2);
XPRBaddqterm(prob,x[1],x[1],1);
```

### 3.5 User error handling

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

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

```c
#include <stdio.h>
#include <setjmp.h>
#include <string.h>
#include "xprb.h"

jmp_buf model_failed; /* Marker for the longjump */
```
void modinf(XPRBprob prob)
{
    XPRBvar x[3];
    XPRBctr ctr[2], cobj;
    int i;

    for(i=0; i<2; i++)    /* Create two integer variables */
        x[i] = XPRBnewvar(prob, XPRB_UI, XPRBnewname("x_%d", i), 0, 100);

    /* Create the constraints: */
    C1: 2x0 + 3x1 >= 41
    C2: x0 + 2x1 = 13 */

    ctr[0] = XPRBnewctr(prob, "C1", XPRB_G);
    XPRBaddterm(ctr[0], x[0], 2);
    XPRBaddterm(ctr[0], x[1], 3);
    XPRBaddterm(ctr[0], NULL, 41);

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

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

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

    XPRBprintprob(prob); /* Print current problem */
    XPRBsolve(prob, ""); /* Solve the LP */
    XPRBprintf(prob, "problem status: %d LP status: %d MIP status: %d
",
                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,\n", XPRBgetvarname(x[i]),
                    XPRBgetsol(x[i]));
    XPRBprintf(prob, "\n");
}

void modinf(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);
}

void modinf(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)
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)
{
    fprintf(stderr,"I cannot create the problem\n");
    return 1;
}
else
{
    fprintf(stderr,"I cannot build the problem\n");
    XPRBdelprob(prob); /* Delete the part of the problem that has been created */
    XPRBdefcberr(prob,NULL,NULL); /* Reset the error callback */
    return 1;
}
else
{
    XPRBdefcberr(prob,usererror,NULL); /* Define the error handling callback */
    modinf(prob); /* Formulate and solve the problem */
    XPRBdefcberr(prob,NULL,NULL); /* Reset the error callback */
    return 0;
}

Since this example defines the printing level and the printing callback function before creating the problem (that is, before BCL is initialized), we pass NULL as first argument.
II. BCL library and class reference
Chapter 4

BCL C library functions

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

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

- XPRBprob: a problem definition;
- XPRBvar: a variable;
- XPRBarrrvar: a one-dimensional array, with elements of type XPRBvar;
- XPRBctr: a constraint;
- XPRBcut: a cut;
- 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 terms to an LP-constraint.

**Synopsis**

```c
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

```
```

to the constraint `ctr1`.

```c
XPRBprob prob;
XPRBctr ctr1;
XPRBarrvar ty1;
double cr[] = {2, 13, 15, 6, 8.5};
...
ty1 = XPRBnewarrvar(prob, 5, XPRB_PL, "array1", 0, 500);
ctr1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBaddarrterm(ctr1, ty1, cr);
```

**Further information**

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

**Related topics**

- XPRBaddterm
- XPRBdelctr
- XPRBdelterm
- XPRBnewctr
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.

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

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

int main(int argc, char **argv)
{
    XPRSprob oprob;
    expl1 = XPRBnewprob("cutexample");
    ...
    XPRBsolve(expl1, "g");
}
```

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

Related topics
XPRBnewcut, XPRBdelcut, XPRBsetcutmode.
**XPRBaddcutterm**

**Purpose**
Add a term to a cut.

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

**Arguments**
- `cut` Reference to a cut as resulting from `XPRBnewcut`.
- `var` Reference to a variable, may be `NULL`.
- `coeff` Value to be added to the coefficient of the variable `var`.

**Return value**
0 if function executed successfully, 1 otherwise.

**Example**
Add the term $5.4 \cdot x_1$ to the cut `cut1`.
```c
XPRBcut cut1;
XPRBvar x1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
x1 = XPRBnewvar(expl1, XPRB_UI, "abc3", 0, 100);
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
XPRBaddcutterm(cut1, x1, 5.4);
```

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

**Related topics**
`XPRBnewcut`, `XPRBaddcutarrterm`, `XPRBdelcutterm`, `XPRBsetcutterm`. 
XPRBaddcutarrterm

Purpose
Add multiple 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 \( \sum_{i=0}^{4} c_i \cdot ty_1 \) to the cut cut1.

```c
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, "array1", 0, 500);
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
XPRBaddcutarrterm(cut1, ty1, cr);
```

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

Related topics
XPRBnewcut, XPRBaddcutterm, XPRBdelcutterm.
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.
XPRBaddqterm

Purpose

Add a quadratic term to the objective function.

Synopsis

```c
int XPRBaddqterm(XPRBprob prob, XPRBvar var1, XPRBvar var2,
                   double coeff);
```

Arguments

- `prob` Reference to a problem.
- `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 objective function:

```c
XPRBprob prob;
XPRBvar x2, x4;
...
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
x4 = XPRBnewvar(prob, XPRB_PL, "abc5", 0, XPRB_INFINITY);
XPRBaddqterm(prob, x2, x4, -2);
```

Further information

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

Related topics

XPRBdelqobj, 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.

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 ,XPRB_INFINITY);
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);
XPRBaddsosel(set1, x2, 9);

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

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

Related topics
XPRBaddsosarrel, XPRBdelsos, XPRBdelsosel, XPRBnewsos.
**XPRBaddterm**

**Purpose**
Add a term to a constraint.

**Synopsis**
```c
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**
```c
    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 $5.4 \times x1$ to the constraint `ctr1`.

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

**Related topics**
- `XPRBaddarrterm`, `XPRBdelctr`, `XPRBdelterm`, `XPRBnewctr`, `XPRBsetter`.

---

*Additional content (if relevant)*

---

*Technical details (if relevant)*

---

*Other related functions (if relevant)*

---

*Notes (if relevant)*

---

*References (if relevant)*

---

*Related libraries (if relevant)*

---

*Additional resources (if relevant)*

---
**XPRBapparrvarel**

**Purpose**
Add an entry to a variable array.

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

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

**Related topics**
- `XPRBdelarrvar`
- `XPRBendarrvar`
- `XPRBnewarrvar`
- `XPRBsetarrvarel`
- `XPRBstartarrvar`
XPRBcleardir

Purpose
Delete all directives.

Synopsis
int XPRBcleardir(XPRBprob prob);

Argument
prob Reference to a problem.

Return value
0 if function executed successfully, 1 otherwise.

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

This deletes all directives for the current problem, expl2.

Related topics
XPRBsetvardir, XPRBsetsosdir.
XPRBdefcbdelvar

Purpose
Callback for interface update at deletion of variables.

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

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

Return value
0 if function executed successfully, 1 otherwise.

Example
Define the variable interface callback function:

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

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

Related topics
XPRBgetvarlink, XPRBsetvarlink.
XPRBdefcberr

Purpose

Callback for user error handling.

Synopsis

```c
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.

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

Further information

1. This function defines the error handling callback that returns the error number and text of error messages and warnings produced by BCL for a given problem. A list of BCL error messages with some explanations can be found in the Appendix A of this manual. If printing of error or warning messages is enabled (see `XPRBsetmsglevel`) these are printed after the call to this function.

2. It is recommended to define this callback function if the error handling by BCL is disabled (for instance in BCL programs integrated into larger applications or in BCL programs executed under Windows). Alternatively it is of course possible to test the return values of all BCL functions. However, the callback provides more detailed information about the type of error that has occurred.

3. This function may be used before any problems have been created and even before BCL has been initialized (with first argument NULL). In this case the error handling function set by this callback applies to all problems that are created subsequently.

Related topics

- `XPRBdefcbmsg, XPRBgetversion, XPRBseterrctrl`
Purpose
Callback for printed output.

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

Further information
1. This function defines a callback function that returns any messages enabled by the setting of `XPRBsetmsglevel`, including warnings and error messages, any other output produced by BCL, and any messages from the Optimizer library. Independent of the message printing settings, this callback also returns output printed by the user’s program with function `XPRBprintf`. If this callback is not defined by the user, any program output is printed to standard output with the exception of warnings and error messages which are printed to the standard error output channel.

2. This function may be used before any problems have been created and even before BCL has been initialized (with first argument NULL). In this case the printing function set by this callback applies to all problems that are created subsequently.

3. A BCL program must not define the message callback `XPRSsetcbmessage` of Xpress-Optimizer (however, all other logging callbacks of the Optimizer may be used).

Related topics
- `XPRBdefcberr`
- `XPRBsetmsglevel`
XPRBdelarrvar

**Purpose**
Delete a variable array.

**Synopsis**
```c
int XPRBdelarrvar(XPRBarrvar av);
```

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

**Return value**
0 if function executed successfully, 1 otherwise.

**Example**
```c
XPRBprob prob;
XPRBarrvar av2;

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

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

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

**Related topics**
- `XPRBapparrvarel`
- `XPRBendarrvar`
- `XPRBnewarrvar`
- `XPRBsetarrvarel`
- `XPRBstartarrvar`
XPRBdelbasis

Purpose
Delete a previously saved basis.

Synopsis
void XPRBdelbasis(XPRBbasis basis);

Argument
basis Reference to a previously saved basis.

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

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

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

**Argument**
`ctr`  
BCL reference to a constraint.

**Return value**
0 if function executed successfully, 1 otherwise.

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

This deletes the constraint `ctrl`.

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

**Related topics**
`XPRBnewctr`. 
**XPRBdelcut**

**Purpose**
Delete a cut definition.

**Synopsis**
```c
int XPRBdelcut(XPRBcut cut);
```

**Argument**
cut  Reference to a cut.

**Return value**
0 if function executed successfully, 1 otherwise.

**Example**
The example shows how to delete cut cut1.

```c
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 $5 \cdot 4 \cdot x_1$ to the cut cut1 and then delete it.

```c
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.
```
XPRBdelqobj

Purpose
Delete the quadratic terms in the objective function.

Synopsis
int XPRBdelqobj(XPRBprob prob);

Argument
prob  Reference to a problem.

Return value
0 if function executed successfully, 1 otherwise.

Example
The following deletes all quadratic objective terms.

    XPRBprob prob;
    XPRBvar x2;
    ...
    x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
    XPRBaddqterm(prob, x2, x2, 5.2);
    XPRBdelqobj(prob);

Further information
This function deletes all quadratic terms from the objective function of the given problem. Note that the quadratic terms are not deleted if the objective function constraint is deleted or changed (using functions XPRBdelctr or XPRBsetobj).

Related topics
XPRBaddqterm, XPRBsetqterm.
XPRBdelsos

Purpose
Delete a SOS.

Synopsis
int XPRBdelsos(XPRBsos sos);

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

Return value
0 if function executed successfully, 1 otherwise.

Example
The following deletes the SOS set1.

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

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

Related topics
XPRBaddsosarrel, XPRBaddsosel, XPRBdelsosel, XPRBnewsos.
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.

```c
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 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 \texttt{x1} from the constraint.

   \begin{verbatim}
   XPRBprob prob;
   XPRBctr ctrl1;
   XPRBvar x1;
   ...
   x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);
   ctrl1 = XPRBnewctr(prob, "r1", XPRB_E);
   XPRBaddterm(ctrl1, x1, 5.4);
   XPRBdelterm(ctrl1, x1);
   \end{verbatim}

Further information
   This function deletes a variable term from the given constraint.

Related topics
   XPRBaddarrterm, XPRBaddterm, XPRBdelctr, XPRBnewctr, XPRBsetterm.
XPRBendarrvar

Purpose
End the definition of a variable array.

Synopsis
int XPRBendarrvar(XPRBarrvar av);

Argument
av  BCL reference to an array.

Return value
0 if function executed successfully, 1 otherwise.

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

This terminates the definition of the array av2.

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

Related topics
XPRBdelarrvar, XPRBnewarrvar, XPRBstartarrvar.
XPRBexportprob

Purpose
Print problem matrix to a file.

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

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

Return value
0 if function executed successfully, 1 otherwise.

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

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

Further information
1. This function prints the matrix to a file with an extended LP or extended MPS format. Provided
   that no extension is given by the user LP files receive the extension .lp. MPS files always
   receive the extension .mat. This function is not available in the student version.

2. When exporting matrices semi-continuous and semi-continuous integer variables are prepro-
   cessed: if a lower bound value greater than 0 is given, then the variable is treated like a
   continuous (resp. integer) variable.

Related topics
XPRBprintprob, XPRBprintf.
**XPRBfinish, XPRBfree**

**Purpose**
Terminate BCL and release system resources.

**Synopsis**

```c
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:

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

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

**Related topics**

XPRBdelprob, XPRBresetprob, XPRBinit.
**XPRBfixvar**

**Purpose**
Fix a variable.

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

**Further information**
This function fixes a variable to the given value. It replaces calls to `XPRBsetub` and `XPRBsetlb`. The value `val` may lie outside the original bounds of the variable.

**Related topics**
- `XPRBgetbounds`, `XPRBgetlim`, `XPRBsetlb`, `XPRBsetlim`, `XPRBsetub`. 
XPRBgetact

Purpose
Get activity value for a constraint.

Synopsis
double XPRBgetact(XPRBctr ctr);

Argument
ctr Reference to a constraint.

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

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

This obtains the activity value for the constraint ctr2.

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

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

Related topics
XPRBgetdual, XPRBgetobjval, XPRBgetrcost, XPRBgetslack, XPRBgetsol, XPRBsync.
**XPRBgetarrvarname**

**Purpose**
Get the name of an array of variables.

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

**Argument**

`av`  
BCL reference to an array of variables.

**Return value**
Name of the array if function executed successfully, NULL otherwise.

**Example**
```
XPRBprob prob;
XPRBarrvar tyl;
...

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

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
**XPRBgetarrvarsiz**

**Purpose**
Get the size of an array of variables.

**Synopsis**
```c
int XPRBgetarrvarsiz(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**
```c
XPRBprob prob;
XPRBarrvar ty1;
int tsize;
...
ty1 = XPRBnewarrvar(prob, 10, XPRB_PL, "array1", 0, 500);
tsize = XPRBgetarrvarsiz(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**

```c
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**

```c
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.

Related topics
XPRBnewname.
**XPRBgetcolnum**

**Purpose**
Get the column number for a variable.

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

**Argument**
- `var`  BCL reference to a variable.

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

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

This gets the column number for variable `x1`.

**Further information**
This function returns the column number of a variable in the matrix currently loaded in the Xpress-Optimizer. If the variable is not part of the matrix, or if the matrix has not yet been generated, the function returns a negative value. To check whether the matrix has been generated, use function `XPRBgetprobstat`. The counting of column numbers starts with 0.

**Related topics**
- `XPRBgetvarname`, `XPRBgetvartype`.
**XPRBgetctrname**

**Purpose**
Get the name of a constraint.

**Synopsis**

```c
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**

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

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;
XPRB_LOACT  lower activity;
XPRB_UUP    upper unit cost;
XPRB_UDN    lower unit cost.

Return value
Ranging information of the required type.

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

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

**Purpose**
Get the row type of a constraint.

**Synopsis**

```c
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`.

```c
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`, `XPRBnewctr`, `XPRBsetctrtype`.
XPRBgetcutid

Purpose
Get the classification or identification number of a cut.

Synopsis
int XPRBgetcutid(XPRBcut cut);

Argument
cut  Reference to a previously created cut.

Return value
Classification or identification number.

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

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

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

Related topics
XPRBnewcut, XPRBgetcuttype, XPRBgetcutrhs, XPRBsetcutid.
**XPRBgetcutrhs**

**Purpose**
Get the RHS value of a cut.

**Synopsis**
```c
double XPRBgetcutrhs(XPRBcut cut);
```

**Argument**
cut  Reference to a previously created cut.

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

**Example**
Get the RHS value of the cut cut1.
```c
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**
```c
int XPRBgetcuttype(XPRBcut cut);
```

**Argument**
cut Reference to a previously created cut.

**Return value**
- `XPRB_L` (inequality)
- `XPRB_G` (inequality)
- `XPRB_E` (equation)
- `-1` An error has occurred,

**Example**
Get the type of cut1.
```c
XPRBcut cut1;
int rtype;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
rtype = XPRBgetcuttype(cut1);
```

**Further information**
This function returns the type of the given cut.

**Related topics**
- `XPRBnewcut`, `XPRBgetcutid`, `XPRBgetcutrhs`, `XPRBsetcuttype`. 
XPRBgetdual

Purpose
Get dual value.

Synopsis
double XPRBgetdual(XPRBctr ctr);

Argument
ctr Reference to a constraint.

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

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

This obtains the dual value for the constraint ctr2.

Further information
This function returns the dual value for a constraint. The user may wish to test first whether this constraint is part of the problem, for instance by checking that the row number is non-negative.
If this function is called after completion of a global search and an integer solution has been found (that is, if function XPRBgetmipstat returns values XPRB_MIP_SOLUTION or XPRB_MIP_OPTIMAL), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the dual value in the last LP that has been solved. If this function is used during the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to XPRBsync with the flag XPRB_XPRS_SOL.

Related topics
XPRBgetactivity, XPRBgetobjval, XPRBgetrcost, XPRBgetslack, XPRBgetsol, XPRBsync.
XPRBgetidxel

Purpose
Get the index number of an index.

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

Arguments
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 i\textsuperscript{th} element in the set if function executed successfully, NULL otherwise.

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

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

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

Related topics
XPRBaddidxel, XPRBgetidxsetname, XPRBgetidxel, XPRBnewidxset.
**XPRBgetidxsetname**

**Purpose**
Get the name of an index set.

**Synopsis**
```c
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).
umiis   Sequence number of the IIS (values <1 access the first set found).

Return value
0 if function executed successfully, 1 otherwise.

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

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

Further information
1. This function returns the variables and constraints forming an IIS (irreducible infeasible set) in an infeasible LP problem. The number of independent IIS identified by Xpress-Optimizer can be obtained with function XPRBgetnumiis.
2. The arrays of variables and constraints that are allocated by this function must be freed by the user’s program.

Related topics
XPRBgetnumiis, XPRBgetlpstat.
XPRBgetlim

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

**Synopsis**
```c
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**
```c
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 matrix is not semi-definite.

Example
The following returns the current LP status.

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

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

Related topics
XPRBgetmipstat, XPRBgetprobstat.
**XPRBgetmipstat**

**Purpose**
Get the MIP status.

**Synopsis**
```c
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.

**Example**
The following returns the current MIP status.
```c
XPRBprob expl2;
int status;
expl2 = XPRBnewprob("example2");
...
XPRBsolve(expl2, "g");
status = XPRBgetmipstat(expl2);
```

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

**Related topics**
- `XPRBgetlpstat`
- `XPRBgetprobstat`
XPRBgetmodcut

Purpose
Get the type of a constraint.

Synopsis
int XPRBgetmodcut(XPRBctr ctr);

Argument
ctr Reference to a previously created constraint.

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

Example
XPRBprob prob;
XPRBctr ctrl1;
int mcstat;
...
ctrl1 = XPRBnewctr(prob, "r1", XPRB_E);
mcstat = XPRBgetmodcut(ctrl1);

This determines whether ctrl1 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**
```c
int XPRBgetnumiis(XPRBprob prob);
```

**Argument**
- `prob`  Reference to a problem.

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

**Example**
The following gets the number of IIS for a problem.
```
XPRBprob expl2;
int num;
expl2 = XPRBnewprob("example2");
...
XPRBsolve(expl2, "");
if(XPRBgetlpstat(expl2)==XPRB_LP_INFEAS)
    num = XPRBgetnumiis(expl2);
```

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

**Related topics**
- `XPRBgetiis`, `XPRBgetlpstat`.
XPRBgetobjval

Purpose
Get the objective function value.

Synopsis
double XPRBgetobjval(XPRBprob prob);

Argument
prob   Reference to a problem.

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

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

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

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

Related topics
XPRBgetdual, XPRBgetrcost, XPRBgetsol, XPRBgetslack, XPRBgetact, XPRBsync.
**XPRBgetprobname**

**Purpose**
Get the name of the specified problem.

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

**Argument**
prob  Reference to a problem.

**Return value**
Name of the problem if function executed successfully, NULL otherwise.

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

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

**Related topics**
XPRBdelprob, XPRBnewname, XPRBnewprob.
XPRBgetprobstat

**Purpose**
Get the problem status.

**Synopsis**
```c
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)
    XPRBsolve(expl2, ")
```

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

**Related topics**

- XPRBgetlpstat
- XPRBgetmipstat
XPRBgetrange

Purpose
Get the range values for a range constraint.

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

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

Argument
```c
var Reference to a variable.
```

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

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

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

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

If this function is called after completion of a global search and an integer solution has been found (that is, if function \( XPRB\text{getmipstat} \) returns values \( XPRB\text{MIP\_SOLUTION} \) or \( XPRB\text{MIP\_OPTIMAL} \)), it returns the value in the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the reduced cost value in the last LP that has been solved. If this function is used during the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to \( XPRB\text{sync} \) with the flag \( XPRB\text{XPRS\_SOL} \).

Related topics
\( XPRB\text{getdual}, XPRB\text{getobjval}, XPRB\text{getslack}, XPRB\text{getsol}, XPRB\text{sync} \).
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.

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

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

Related topics
XPRBaddterm, XPRBgetctrtype, XPRBsetctrtype, XPRBsetterm.
**XPRBgetrownum**

**Purpose**
Get the row number for a constraint.

**Synopsis**
```c
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`.
```c
XPRBprob prob;
XPRBctr ctrl;
...
int rindex;
ctrl = XPRBnewctr(prob, "r1", XPRB_E);
rindex = XPRBgetrownum(ctrl);
```

**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 functions returns the objective sense (maximization or minimization). The sense is set to
minimization by default and may be changed with functions XPRBsetmax, XPRBminim, and
XPRBmaxim.

Related topics
XPRBmaxim, XPRBminim, XPRBsetmax, XPRBsolve.
**XPRBgetslack**

**Purpose**
Get slack value for a constraint.

**Synopsis**
```c
double XPRBgetslack(XPRBctr ctr);
```

**Argument**
- `ctr`: Reference to a constraint.

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

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

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

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

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

**Related topics**
- `XPRBgetact`
- `XPRBgetdual`
- `XPRBgetobjval`
- `XPRBgetrcost`
- `XPRBgetsol`
- `XPRBsync`
**XPRBgetsol**

**Purpose**
Get solution value for a variable.

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

**Argument**
- `var` Reference to a variable.

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

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

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

**Further information**
1. This function returns the current solution value for a variable. The user may wish to test first whether this variable is part of the problem, for instance by checking that the column number is non-negative.
   If this function is called after completion of a global search and an integer solution has been found (that is, if function `XPRBgetmipstat` returns values `XPRB_MIP_SOLUTION` or `XPRB_MIP_OPTIMAL`), it returns the value of the best integer solution. If no integer solution is available after a global search this function outputs a warning and returns 0. In all other cases it returns the solution value in the last LP that has been solved. If this function is used during the execution of an optimization process (for instance in Optimizer library callback functions) it needs to be preceded by a call to `XPRBsync` with the flag `XPRB_XPRS_SOL`.

2. Note that “integer solution” means “solution within the integer feasibility limits”, that means for any comparison of solution values the current Optimizer tolerance settings have to be taken into account. So care must be taken when handling the solution values of integer variables. For example, you cannot simply treat the value as an integer, because a value such as 0.999998, may well be truncated to zero. Instead, you must make sure you round the value to the nearest integer.

**Related topics**
- `XPRBgetact`, `XPRBgetdual`, `XPRBgetobjval`, `XPRBgetrcost`, `XPRBgetslack`, `XPRBsync`. 
**XPRBgetsosname**

**Purpose**
Get the name of a SOS.

**Synopsis**
```c
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**
```c
XPRBprob prob;
XPRBsos set1;
...
set1 = XPRBnewsos(prob, "sos1", XPRB_S1);
printf("%s
", 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

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

Return value
System time measure in milliseconds.

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

```c
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**
```c
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`:
```c
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**
```c
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.

```c
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**

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

**Arguments**

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

**Return value**
Ranging information of the required type.

**Example**

This example retrieves the upper cost value for a variable.

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

**Further information**

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

**Related topics**

- `XPRBnewvar, XPRBgetctrrng`
**XPRBgetvartype**

**Purpose**
Get the type of a variable.

**Synopsis**
```c
int XPRBgetvartype(XPRBvar var);
```

**Argument**
```c
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**

```c
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:

```c
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`, Appendix B.
**XPRBinit**

**Purpose**

Initialize BCL.

**Synopsis**

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

**Return value**

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

**Example**

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

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

**Further information**

1. This function explicitly initializes BCL, that is it tests whether a license for running this software is available. It is possible to run BCL with a student license; this mode implies restrictions to the available functionality and to the accepted problem size.

2. The initialization is also performed by function XPRBnewprob so that usually there is no need to call this explicit initialization. This function may be used if the embedding of BCL into some larger application requires a test of the license at an earlier stage, before even creating any model. Note that this function also initializes Xpress-Optimizer, so that it is usually not necessary to call XPRSinit separately (the latter is only required if one wishes to continue using the optimizer after terminating BCL).

**Related topics**

XPRBfree, XPRBnewprob, XPRSinit (see Optimizer Reference Manual).
XPRBloadbasis

Purpose
Load a previously saved basis.

Synopsis
```c
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.

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

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

Related topics
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.
Purpose
Maximize the objective function for the active problem.

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

Arguments
prob Reference to a problem.
flags Choice of the solution algorithm, which may be one of:
" " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);
"d" solve the problem using the dual simplex algorithm;
"p" solve the problem using the primal simplex algorithm;
"b" solve the problem using the Newton barrier algorithm;
"n" use the network solver (LP only);
"l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);
"g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value
0 if function executed successfully, 1 otherwise.

Example
Maximize the LP problem using a Newton-Barrier algorithm.

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

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

Related topics
XPRBgetobjval, XPRBgetsol, XPRBminim, XPRBsetsense, XPRSmaxim (see Optimizer Reference Manual).
**XPRBminim**

**Purpose**
Minimize the objective function for the active problem.

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

**Arguments**
- `prob` Reference to a problem.
- `flags` Choice of the solution algorithm, which may be one of:
  - " " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);
  - "d" solve the problem using the dual simplex algorithm;
  - "p" solve the problem using the primal simplex algorithm;
  - "b" solve the problem using the Newton barrier algorithm;
  - "n" use the network solver (LP only);
  - "l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);
  - "g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

**Return value**
0 if function executed successfully, 1 otherwise.

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

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

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

**Related topics**
- `XPRBgetobjval`, `XPRBgetsol`, `XPRBmaxim`, `XPRBsetobj`, `XPRBsolve`, `XPRBsync`, `XPRSminim` (see Optimizer Reference Manual).
**XPRBnewarrsum**

**Purpose**
Create a sum constraint with individual coefficients.

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

**Arguments**
- **prob**: Reference to a problem.
- **name**: The constraint name (of unlimited length). May be `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 \( \sum_{i=0,...,4} c[i] \cdot t[i] \geq 7.0 \).
```
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 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`.

**Related topics**
`XPRBdelctr, XPRBnewctr, XPRBnewprec, XPRBnewsum`. 
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.

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.
**XPRBnewctr**

**Purpose**
Create a new constraint.

**Synopsis**

```c
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.

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

**Further information**
This function creates a new constraint and returns the reference to this constraint, i.e. the constraint’s model name. It has to be called before **XPRBaddterm** 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**.

**Related topics**

- **XPRBaddterm**, **XPRBdelctr**, **XPRBdelterm**.
XPRBnewcut

Purpose
Create a new cut.

Synopsis
XPRBcut XPRBnewcut(XPRBprob prob, char qrtype, int mtype);

Arguments
prob  Reference to a problem.
qrtype  Type of the cut:
            XPRB_L  ≤ (inequality)
            XPRB_G  ≥ (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, XPRBaddcarts.
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**.

**Related topics**
**XPRBnewcut**, **XPRBaddcutterm**.
**XPRBnewcutprec**

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

**Synopsis**

\[\text{XPRBcut XPRBnewcutprec(XPRBprob prob, XPRBvar v1, double dur, XPRBvar v2, int mtype);}\]

**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 \(ty_1^2 + 5.4 \leq ty_1^4\).

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

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

**Related topics**
**XPRBnewcut, XPRBaddcutterm.**
XPRBnewcutsum

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

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

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

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

Example
Create the equality constraint \( \sum_{i=1}^{4} y_1 = 9 \).

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

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

Related topics
XPRBnewcut, XPRBaddcutterm.
XPRBnewidxset

**Purpose**
Create a new index set.

**Synopsis**

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

**Arguments**

- `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`.

**Related topics**

- `XPRBaddidxel`
- `XPRBgetidxel`
- `XPRBgetidxsetname`
- `XPRBgetidxsetsize`
**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 formatting 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`
- `XPRBgetprobrname`
- `XPRBnewprob`
XPRBnewprec

Purpose
Create a precedence constraint $v_1 + \text{dur} \leq v_2$.

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 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 withCTR.

Related topics
XPRBnewarrsum, XPRBnewsum.
**XPRBnewprob**

**Purpose**
Initialize a new problem.

**Synopsis**

```c
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`.

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

**Further information**

1. This function needs to be called to create and initialize a new problem. This function initializes BCL and also Xpress-Optimizer; it is **not necessary** to call `XPRSinit` from the user’s program. If the initialization does not find a valid license, BCL does not initialize. It is possible to run BCL with a student license; this mode implies restrictions to the available functionality and to the accepted problem size.

2. The name given to a problem determines the name and the location of the working files of Xpress-Optimizer. At the creation of a problem any existing working files of the same name are deleted. When solving several instances of a problem simultaneously the user must make sure to assign a different name to every instance. If no problem name is indicated, BCL creates a unique name including the full path to the temporary directory (Xpress-Optimizer creates its working files in the temporary directory).

**Related topics**

- `XPRBdelprob`
- `XPRBgetprobnamex`
- `XPRBinit`
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.

Related topics
XPRBdelsos, XPRBgetsosname, XPRBgetsostype, XPRBnewsosrc, XPRBnewsosw.
**XPRBnewsosrc**

**Purpose**
Create a SOS, using a reference constraint.

**Synopsis**
```c
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 if not required.

**Return value**
Reference to the new SOS if function executed successfully, NULL otherwise.

**Example**
The following creates an SOS of type 2 with variables from the array `ty1`, and their coefficients in the constraint `ctr4`.
```c
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, "array1", 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.

**Related topics**
- XPRBdelsos, XPRBgetsosname, XPRBgetsostype, XPRBnewsos, XPRBnewsosw.
**XPRBnewsosw**

**Purpose**  
Create a SOS, using weights.

**Synopsis**  
```c
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`.

```c
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`.

**Related topics**  
- `XPRBdelsos`
- `XPRBgetsosname`
- `XPRBnewsos`
- `XPRBnewsosrc`
XPRBnewsum

Purpose
Create a sum constraint.

Synopsis
XPRBctr XPRBnewsum(XPRBprob prob, const char *name, XPRBarrvar av,
   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.
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 \( \sum_{i=0:4} tyl[i] = 9 \).

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

Related topics
XPRBnewarrsum, XPRBnewctr, XPRBnewprec.
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-MP constants). The function returns the
BCL reference to the variable (i.e. a model variable). If the indicated name is already in use, BCL
adds an index to it. If no variable name is given, BCL generates a default name starting with
VAR. 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.
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 tyl;
...
tyl = XPRBnewarrvar(prob, 5, XPRB_PL, "arry1", 0, 500);
XPRBprintarrvar(tyl);

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

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctr</td>
<td>Reference to a constraint.</td>
</tr>
</tbody>
</table>

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

Argument
```c
cut   Reference to a cut.
```

Return value
0 if function executed successfully, 1 otherwise.

Example
Print out the cut cut2.
```c
XPRBcut cut2;
XPRBarrvar ty1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
ty1 = XPRBnewarrvar(expl1, 5, XPRB_PL, "arry1", 0, 500);
cut2 = XPRBnewcutsum(expl1, ty1, XPRB_E, 9, 3);
XPRBprintcut(cut2);
```

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

Related topics
```c
XPRBnewcut.
```
XPRBprintf

Purpose
Print text and other program output.

Synopsis
```c
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.

```c
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
XPRBprintfprob, 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.
XPRBprintprob

Purpose
Print out the specified problem.

Synopsis
int XPRBprintprob(XPRBprob prob);

Argument
prob Reference to a problem.

Return value
0 if function executed successfully, 1 otherwise.

Example
The following prints out the current problem definition.

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

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

Related topics
XPRBexportprob, XPRBprintf.
**XPRBprintsos**

**Purpose**
Print out a Special Ordered Set.

**Synopsis**
```c
int XPRBprintsos(XPRBsos sos);
```

**Argument**
sos Reference to a Special Ordered Set.

**Return value**
0 if function executed successfully, 1 otherwise.

**Example**
```c
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.
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].

```c
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
```c
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
```c
double vlist[10];
FILE *datafile;
...
datafile=fopen("filename", "r");
XPRBreadarrlinecb(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
- `XPRBreadarrlinecb`, `XPRBsetdecsign`.
## XPRBreadlinecb

### Purpose
Read a fixed-format line from a data file.

### Synopsis
```c
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.

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

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

### Related topics
- `XPRBreadarrlinecb`, `XPRBsetdecsign`.
**XPRBresetprob**

**Purpose**
Release system resources used for storing solution information.

**Synopsis**
```c
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:

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

**Further information**
This function deletes any solution information stored in BCL; it also deletes the corresponding Xpress-Optimizer problem and removes any auxiliary files that may have been created by optimization runs. It also resets the Optimizer control parameters for spare matrix elements (EXTRACOLS, EXTRAROWS, and EXTRAELEMENTS) 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**

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

**Argument**

- `prob`  Reference to a problem.

**Return value**
Reference to the saved basis.

**Example**

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

This saves the current basis.

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

**Related topics**

- `XPRBdelbasis`, `XPRBloadbasis`, `XPRSreadbasis` (see Optimizer Reference Manual),
- `XPRSwritebasis` (see Optimizer Reference Manual).
**XPRBsetarrvarel**

**Purpose**
Add an entry to a variable array in a given position.

**Synopsis**
```
int XPRBsetarrvarel(XPRBarrvar av, int ndx, XPRBvar var);
```

**Arguments**
- **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.

**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**

```c
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**

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

This changes `ctrl` to a ‘less than or equal to’ constraint.

**Further information**

This function changes the type of a previously defined constraint to inequality, equation or non-binding. Function `XPRBsetrange` has to be used for changing the constraint to a ranged constraint. If a ranged constraint is changed back to some other type with this function, its upper bound becomes the right hand side value.

**Related topics**

- `XPRBgetctrtype`,
- `XPRBnewctr`,
- `XPRBsetrange`,
- `XPRBsetterm`.
XPRBsetcutid

Purpose
Set the classification or identification number of a cut.

Synopsis
int XPRBsetcutid(XPRBcut cut, int id);

Arguments
- 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 cut1 to 10.

XPRBcut cut1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
XPRBsetcutid(cut1, 10);

Further information
This function changes the classification or identification number of a previously defined cut. This change does not have any effect on the cut definition in Xpress-Optimizer if the cut has already been added to the matrix with the function XPRBaddcuts.

Related topics
XPRBnewcut, XPRBgetcutid, XPRBsetcuttype.
XPRBsetcutmode

Purpose
Set the cut mode.

Synopsis

int XPRBsetcutmode(XPRBprob prob, int mode);

Arguments
prob     Reference to a problem.
mode     Cut mode indicator:
          0    switch cut mode off
          1    switch cut mode on

Return value
0 if function executed successfully, 1 otherwise.

Example
The example shows how to enable the cut mode.

XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
XPRBsetcutmode(expl1, 1);

Further information
This function switches the cut mode on or off. It changes the settings of certain Optimizer
controls. Switching the cut mode off resets these controls to their default values.

Related topics
XPRBaddcuts.
XPRBsetcutterm

Purpose
Set a cut term.

Synopsis

    int XPRBsetcutterm(XPRBcut cut, XPRBvar var, double coeff);

Arguments

    cut     Reference to a previously created cut.
    var     Reference to a variable, may be NULL.
    coeff   Value of the coefficient of the variable var.

Return value
0 if function executed successfully, 1 otherwise.

Example
Set the RHS of the cut cut1 to 7.0.

    XPRBcut cut1;
    XPRBprob expl1;
    expl1 = XPRBnewprob("cutexample");
    cut1 = XPRBnewcut(expl1, XPRB_E, 1);
    XPRBsetcutterm(cut1, NULL, 7.0);

Further information
This function sets the coefficient of a variable to the value coeff. If var is set to NULL, the right hand side of the cut is set to coeff.

Related topics
XPRBnewcut, XPRBaddcutterm, XPRBdelcutterm.
XPRBsetcuttype

Purpose
Set the type of a cut.

Synopsis
int XPRBsetcuttype(XPRBcut cut, int type);

Arguments
- cut: Reference to a previously created cut.
- type: Type of the cut:
  - XPRB_L \leq \text{(inequality)}
  - XPRB_G \geq \text{(inequality)}
  - XPRB_E = \text{(equation)}

Return value
0 if function executed successfully, 1 otherwise.

Example
Set the type of cut1 to ‘\leq’.
XPRBcut cut1;
XPRBprob expl1;
expl1 = XPRBnewprob("cutexample");
cut1 = XPRBnewcut(expl1, XPRB_E, 1);
XPRBsetcuttype(cut1, XPRB_L);

Further information
This function changes the type of the given cut. This change does not have any effect on the cut definition in Xpress-Optimizer if the cut has already been added to the matrix with the function XPRBaddcuts.

Related topics
- XPRBnewcut, XPRBgetcuttype, XPRBgetcutid.
XPRBsetdecsign

Purpose
Select the decimal sign for data input.

Synopsis
```c
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
```c
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 `XPRBreadarrlinecb` and `XPRBreadlinecb` can be changed to a comma or any other non-numerical ASCII character. Note that all output still contains the decimal point.

Related topics
- `XPRBreadarrlinecb`, `XPRBreadlinecb`.
XPRBsetdictionarysize

Purpose
Set the size of a dictionary.

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

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

Return value
0 if function executed successfully, 1 otherwise.

Example
Switch off the names dictionary:

  XPRBprob expl2;
  expl2 = XPRBnewprob("example2");
  XPRBsetdictsize(expl2, XPRB_DICT_NAMES, 0);

Further information
This function sets the size of the hash table of the names or indices dictionaries (defaults: names 2999, indices 1009) of the given problem. It can only be called immediately after the creation of the corresponding problem. The names dictionary serves for storing and accessing the names of all modeling objects (variables, arrays of variables, constraints, SOS, index sets), the indices dictionary for all index set elements. Once the names 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. 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 han-
   dling by BCL if a BCL program is embedded into some larger application or executed under
   Windows. Callback function XPRBdefcberr can be defined to retrieve the error messages and
   implement user error handling.

2. This function can be used before BCL has been initialized.

Related topics
XPRBdefcberr, XPRBgetversion.
**XPRBsetlb**

**Purpose**
Set a lower bound.

**Synopsis**
```c
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.
```c
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 $0 \leq x3 \leq 50$ at the creation of this variable to $x3 = 0 \text{ or } 5 \leq x3 \leq 50$.

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**
This function changes the type of a previously defined constraint from ordinary constraint to model cut and vice versa.

**Related topics**
- XPRBdelctr, XPRBgetmodcut, XPRBnewctr.
XPRBsetmsglevel

Purpose
Set the message print level.

Synopsis
int XPRBsetmsglevel(XPRBprob prob, int level);

Arguments
prob  Reference to a problem.
level  The message level, i.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 function 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, but any other type of constraint
may be chosen too. In the latter case, the equation or inequality expressed by the constraint
also remains part of the problem.

Related topics
XPRBgetsense, XPRBsetsense.
**XPRBsetqterm**

**Purpose**
Set a quadratic objective term.

**Synopsis**
```
int XPRBsetqterm(XPRBprob prob, XPRBvar var1, XPRBvar var2,
                 double coeff);
```

**Arguments**
- **prob** Reference to a problem.
- **var1** Reference to a variable.
- **var2** Reference to a variable (not necessarily different).
- **coeff** Value to be added to the coefficient of the term \( var1 \times var2 \).

**Return value**
0 if function executed successfully, 1 otherwise.

**Example**
```
XPRBprob prob;
XPRBvar x2;
...
x2 = XPRBnewvar(prob, XPRB_PL, "abc1", 0, XPRB_INFINITY);
XPRBaddqterm(prob, x2, x2, 1);
XPRBsetqterm(prob, x2, x2, 5.2);
```

This sets the coefficient of the objective term \( x2^2 \) to 5.2.

**Further information**
This function sets the coefficient of a quadratic term in the objective function to the value \( \text{coeff} \).

**Related topics**
- XPRBaddqterm, XPRBdelqobj.
**XPRBsetrange**

**Purpose**
Define a range constraint.

**Synopsis**

```c
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 $4.0 \leq \sum_{i=0:4} ty1[i] \leq 15.5$.

```c
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**
```c
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**
```c
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-MP. This may be a priority for branching on a SOS (type XPRB_PR), the preferred branching direction (types XPRB_UP, XPRB_DN) or the estimated cost incurred when branching on a SOS (types XPRB_PU, XPRB_PD).
Several directives of different types may be set for a single set. Function XPRBsetvardir may be used to set a directive for a variable.

Related topics
XPRBcleardir, XPRBsetvardir.
XPRBsetterm

Purpose
Set a 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 ctrl1;
...
ctrl1 = XPRBnewctr(prob, "r1", XPRB_E);
XPRBsetterm(ctrl1, NULL, 7.0);

This sets the right hand side of the constraint ctrl1 to 7.0.

Further information
This function sets the coefficient of a variable to the value coeff. If var is set to NULL, the right hand side of the constraint is set to coeff.

Related topics
XPRBaddterm, XPRBdelctr, XPRBnewctr.
XPRBsetub

Purpose
Set an upper bound.

Synopsis
int XPRBsetub(XPRBvar var, double bdu);

Arguments
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 $x_1$ 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**
```c
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.

```c
XPRBprob prob;
XPRBvar x1;
...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);
XPRBsetvardir(x1, XPRB_PR, 10);
XPRBsetvardir(x1, XPRB_UP, 0);
```

**Further information**
1. This function sets any type of branching directive available in Xpress-MP. This may be a priority for branching on a variable (type `XPRB_PR`), the preferred branching direction (types `XPRB_UP`, `XPRB_DN`) or the estimated cost incurred when branching on a variable (types `XPRB_PU`, `XPRB_PD`). Several directives of different types may be set for a single variable.

2. Note that it is only 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.

```c
XPRBprob prob;
XPRBvar x1;
...
x1 = XPRBnewvar(prob, XPRB_UI, "abc3", 0, 100);
XPRBsetvartype(x1, XPRB_BV);
```

Further information
This function changes the type of a variable that has been created previously.

Related topics
XPRBgetvarname, XPRBgetvartype, XPRBnewvar.
XPRBsolve

Purpose
Call the Xpress-Optimizer solution algorithm.

Synopsis
int XPRBsolve(XPRBprob prob, char *alg);

Arguments
- prob Reference to a problem.
- alg Choice of the solution algorithm, which should be one of:
  - " " solve the problem using the recommended LP/QP algorithm (MIP problems remain in presolved state);
  - "d" solve the problem using the dual simplex algorithm;
  - "p" solve the problem using the primal simplex algorithm;
  - "b" solve the problem using the Newton barrier algorithm;
  - "n" use the network solver (LP only);
  - "l" relax all global entities (integer variables etc) in a MIP/MIQP problem and solve it as a LP problem (problem is postsolved);
  - "g" solve the problem using the MIP/MIQP algorithm. If a MIP/MIQP problem is solved without this flag, only the initial LP/QP problem will be solved.

Return value
0 if function executed successfully, 1 otherwise.

Example
The following code uses the primal simplex algorithm to solve expl2 as a MIP, assuming that it contains global entities.

```c
XPRBprob expl2;
expl2 = XPRBnewprob("example2");
...
XPRBsolve(expl2, "pg");
```

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

Related topics
XPRBget sense, XPRBmaxim, XPRBminim, XPRBsetsense, XPRBsync.
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 solution currently held in the Optimizer;
- XPRB_XPRS_PROB force problem reloading.

Return value
0 if function executed successfully, 1 otherwise.

Example
The following forces BCL to reload the matrix into the Optimizer even if there has been no change other than bound changes to the problem definition in BCL since the preceding optimization:

```c
XPRBprob expl2;
expl2 = XPRBnewprob("example2");
...
XPRBsolve(expl2, "l");
...
XPRBsync(expl2, XPRB_XPRS_PROB);
XPRBsolve(expl2, "g");
```

Further information
1. This method resets the BCL problem status.
2. XPRB_XPRS_SOL: at the next solution access the solution information in BCL is updated with the solution held in the Optimizer (after MIP search: best integer solution, otherwise solution of the last LP solved).
3. XPRB_XPRS_PROB: at the next call to optimization or XPRBloadmat the problem is completely reloaded into the Optimizer; bound changes are not passed on to the problem loaded in the Optimizer any longer.

Related topics
XPRBgetsol, XPRBgetcost, XPRBgetdual, XPRBgetslack, XPRBloadmat, XPRBminim, XPRBmaxim, XPRBsolve.
**XPRBwritedir**

**Purpose**
Write directives to a file.

**Synopsis**
```c
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:
```c
XPRBprob expl2;
expl2 = XPRBnewprob("example2");
...
XPRBwritedir(expl2, NULL);
```

**Further information**
This function writes out to a file the directives defined for a problem. The extension .dir is appended to the given file name. When no file name is given, the name of the problem is used. If a file of the given name exists already it is replaced.

**Related topics**
XPRBexportprob, XPRBsetvardir, XPRBsetsosdir.
Chapter 5
BCL in C++ and Java

5.1 An overview of BCL in C++

The C++ interface of BCL provides the full functionality of the C version except for the data input, output and error handling for which the corresponding C functions may be used. The C modeling objects, such as variables, constraints and problems, are converted into classes, and their associated functions into methods of the corresponding class in C++.

To use the C++ version of BCL, the C++ header file must be included at the beginning of the program (and not the main BCL header file xprb.h).

```
#include "xprb_cpp.h"
```

Using C++, the termwise definition of constraints is even easier. This has been achieved by overloading the algebraic operators like '+', '-', '<=', or '=='. With these operators constraints may be written in a form that is close to an algebraic formulation.

It should be noted that the names of classes and methods have been adapted to C++ naming standards: All C++ classes that have a direct correspondence with modeling objects in BCL (namely XPRBprob, XPRBvar, XPRBctr, XPRBcut, XPRBsos, XPRBindexSet, XPRBbasis) take the same names, with the exception of XPRBindexSet. In the names of the methods the prefix XPRB has been dropped, as have been references to the type of the object. For example, function XPRBgetvarname is turned into the method getName of class XPRBvar.

All C++ classes of BCL are part of the namespace dashoptimization. To use the (short) class names, it is recommended to add the line

```
using namespace ::dashoptimization;
```

at the beginning of every program that uses the C++ classes of BCL.

C++ functions can be used together with C functions, for instance when printing program output or using Xpress-Optimizer functions. However, it is not possible to mix BCL C and C++ objects in a program.

5.1.1 Example

An example of use of BCL in C++ is the following, which constructs the scheduling example described in Chapter 2:

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

using namespace std;
using namespace ::dashoptimization;

#define NJ 4 // Number of jobs
#define NT 10 // Time limit
```
double DUR[] = {3,4,2,2}; // Durations of jobs
XPRBVar start[NJ]; // Start times of jobs
XPRBVar delta[NJ][NT]; // Binaries for start times
XPRBVar z; // Max. completion time
XPRBprob p("Jobs"); // Initialize BCL & a new problem

void jobsModel()
{
    XPRBlinExp 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);
    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.solve("g"); // Solve the problem as MIP
    statmip = p.getMIPStat(); // Get the MIP problem status
    if((statmip == XPRB_MIP_SOLUTION) ||
        (statmip == XPRB_MIP_OPTIMAL))
    {
        cout << "Objective: " << p.getObjVal() << endl;
        for(j=0;j<NJ;j++)
        {
            cout << start[j].getName() << ": " << start[j].getSol() << endl;
        }
    }
}

int main(int argc, char **argv)
{
    jobsModel(); // Problem definition
    jobsSolve(); // Solve and print solution
}
The definition of SOS is similar to the definition of constraints.

```c
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(XPRBasename("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("sos%d",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.

```c
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.

```c
#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);
```
datafile=fopen("durations.dat","r");
while(NJ<MAXNJ &&
    XPRB_readlinecb(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()
{
    XPRBlinExp 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;
}
5.2 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`, `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

```java
import com.dashoptimization.*;
```

at the beginning of every program that uses the Java classes of BCL.

The overview on C++ and Java classes in section 5.3 gives preference to the C++ notation. A comprehensive documentation of the BCL Java interface is available as a separate ‘Java on-line documentation’.

5.2.1 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`.

```java
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 time variables */
    static XPRBvar[][] delta; /* Binaries for start times */
    static XPRBvar z; /* Max. completion time */
    static XPRB bcl;
    static XPRBprob p;
    static void jobsModel()
    {
        XPRBlinExp 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]; /* Binaries for each job */
        for(j=0;j<NJ;j++)
            for(t=0;t<(NT-DUR[j]+1);t++)
```

BCL in C++ and Java 161 BCL Reference Manual
The definition of SOS is similar to the definition of constraints.

```
delta[j][t] = p.newVar("delta"+(j+1)+(t+1), XPRB.BV);
for(j=0; j<NJ; j++)  /* Calculate max. completion time */
    p.newCtr("Makespan", start[j].add(DUR[j]).leq(z) );

p.newCtr("Prec", start[0].add(DUR[0]).leq(start[2]) );
    /* Precedence rel. between jobs */
for(j=0; j<NJ; j++)  /* Linking start times & binaries */
{
    le = new XPRBlinExp();
    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 XPRBlinExp();
    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.solve("g");  /* Solve the problem as MIP */
    statmip = p.getMIPStat();  /* Get the MIP problem status */
    if((statmip == XPRB.MIP_SOLUTION) ||
      (statmip == XPRB.MIP_OPTIMAL))
    {
      /* An integer solution has been found */
      System.out.println("Objective: "+ p.getObjVal());
      /* Print solution for all start times */
      for(j=0; j<NJ; j++)
        System.out.println(start[j].getName() + ": "+
                          start[j].getSol());
    }
}
```

```
public static void main(String[] args)
{
    bcl = new XPRB();  /* Initialize BCL */
    p = bcl.newProb("Jobs");  /* Create a new problem */
    jobsModel();  /* Problem definition */
    jobsSolve();  /* Solve and print solution */
}
```
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 XPRBlinExp();
    for (t = 0; t < (NT - DUR[j] + 1); t++)
        le.add(delta[j][t].mul((t + 1)));
    set[j] = p.newSos("sosj", XPRB.S1, le);
}

Branching directives for the SOSs are added as follows.

    for (j = 0; j < NJ; j++) set[j].setDir(XPRB.DN);
    /* First branch downwards on sets */

Adding the following two lines during or after the problem definition will print the problem
to the standard output and export the matrix to a file respectively.

    p.print(); /* Print out the problem def. */
    p.exportProb(XPRB.MPS, "expl1"); /* Output matrix to MPS file */

Similarly to what has been shown for the problem formulation in C and C++, we may read data
from file and use index sets in the problem formulation. Only a few changes and additions to
the basic model formulation are required for the creation and use of index sets. However, if
we want to read in a data file in the format accepted by the C functions XPRBreadline and
XPRBreadarrline (that is, using ‘!’ as commentary sign, and ‘,’ as separators, and skip blanks
and empty lines), we need to configure the data file access in Java.

In the following program listing we leave out the method jobsSolve because it remains un-
changed from the previous.

```java
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 = 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 {
        /***** Initialize the stream tokenizer *****/
        initST(file)
        /***** Read data from files *****/
    }
}
```java
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()
{
    XPRBlinExp 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 XPRBlinExp();
        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 */
    { /* Define and set objective */
        le = new XPRBlinExp();
        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);
}
public static void main(String[] args)
{
    bcl = new XPRB();
    p = bcl.newProb("Jobs");
    try
```

BCL in C++ and Java 164 BCL Reference Manual
5.2.2 Error handling

If an error occurs, BCL Java raises exceptions. A large majority of these exceptions are of class XPRBError, during initialization of class XPRBlicenseError, and if file access is involved (such as in method exportProb) of class IOException. For simplicity's sake most of the Java program examples in this manual omit the error handling. Below we show a Java implementation of the example of user error handling with BCL from Section 3.5. Other features demonstrated by this example include

- redirection of the BCL output stream for the whole program and for an individual problem;
- setting the BCL message printing level;
- forcing garbage collection for a problem.

```java
import java.io.*;
import com.dashoptimization.*;

public class xbexpl3 {
    static XPRB bcl;

    public static void modexpl3(XPRBprob prob) throws XPRBerror {
        XPRBvar[] x;
        XPRBlinExp 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 XPRBlinExp();
        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 */

        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;
    XPRB 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 */
        modexp13(prob); /* Formulate and solve the problem */
        prob.setOutputStream(f); /* Redirect problem output to file */
        prob.print(); /* Write to the output file */
        f.close();
        prob=null; /* Delete the problem */
        System.gc(); /* Force garbage collection */
        System.runFinalization();
        System.err.flush();
    }
    catch(IOException e)
    {
        System.err.println(e.getMessage());
        System.exit(1);  
    }
    catch(XPRBerror e)
    {
        System.err.println("BCL error "+ e.getErrorCode() + ": " + e.getMessage());
        System.exit(1);
    }
}

5.3 C++ and Java class reference

The complete set of classes of the BCL C++ and Java interfaces is summarized in the following
list:

XPRB Initialization and general settings (C++ and Java), definition of all parameters (Java only).

p. 184
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.

5.3.1 Classes corresponding to BCL modeling objects

All C++ and Java classes that have a direct correspondence with modeling objects in BCL (namely XPRBprob, XPRBvar, XPRBctr, XPRBcut, XPRBsos, XPRBindexSet, XPRBbasis) take the same names, with the exception of XPRBindexSet. In C++, 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++ or Java problem by using method getXPRSprob of class XPRBprob.

Most of the methods of the classes listed in this section call standard BCL C functions, as indicated, and return their result. These methods are common to C++ and Java and are listed only once, giving preference to the C++ notation (the type char or char * in C++ is replaced by String in Java, and the type bool in C++ becomes boolean in Java, furthermore, in Java there are no pointers). Where the C and 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.

The major difference between the C++ and Java implementations 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. In Java, it is not possible to overload operators; 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. The corresponding operators and methods are presented in separate paragraphs.
XPRBprob

Description
Problem definition, including methods for creating and deleting the modeling objects, problem solving, changing settings, and retrieving solution information.

Constructors
XPRBprob()
Calls XPRBnewprob

XPRBprob(const char *name)
Calls XPRBnewprob

Methods
xbprob *getCRef() (C++ only)
XPRSprob getXPRSprob()
Calls XPRBgetXPRSprob

int reset()
Calls XPRBresetprob

int setColOrder(int num)
Calls XPRBsetcolorder

int setDictionarySize(int dict, int size)
Calls XPRBsetdictionarysize

int setMsgLevel(int lev)
Calls XPRBsetmsglevel

int setRealFmt(const char *fmt)
Calls XPRBsetrealfmt

void setOutputStream(Object stream) (Java only)

void setOutputStream() (Java only)

XPRBvar newVar(const char *name, int type, double lob, double upb)
Calls XPRBnewvar

XPRBvar newVar(const char *name, int type)
Calls XPRBnewvar

XPRBvar newVar(const char *name)
Calls XPRBnewvar

XPRBvar newVar()
Calls XPRBnewvar

XPRBctr newCtr(const char *name, XPRBlinRel& ac)
Calls XPRBnewctr

XPRBctr newCtr(const char *name)
Calls XPRBnewctr

XPRBctr newCtr()
Calls XPRBnewctr

XPRBctr newCtr(XPRBlinRel& ac)
Calls XPRBnewctr

void delCtr(XPRBctr& ctr)
Calls XPRBdelctr

int setObj(XPRBctr ctr)
Calls XPRBsetobj

int setObj(XPRBquadExp q)
int setObj(XPRBlinExp l)
int setObj(XPRBvar v)

void delQObj()
Calls XPRBdelqobj
XPRBsos newSos(int type)
   Calls XPRBnewsos
XPRBsos newSos(const char *name, int type)
   Calls XPRBnewsos
XPRBsos newSos(int type, XPRBlinExp& l)
   Calls XPRBnewsos
XPRBsos newSos(const char *name, int type, XPRBlinExp& l)
   Calls XPRBnewsos

void delSos(XPRBsos& sos)
   Calls XPRBdelsos
XPRBindexSet newIndexSet()
   Calls XPRBnewidxset
XPRBindexSet newIndexSet(const char *name)
   Calls XPRBnewidxset
XPRBindexSet newIndexSet(const char *name, int maxsize)
   Calls XPRBnewidxset
XPRBbasis saveBasis()
   Calls XPRBsavebasis
int loadBasis(const XPRBbasis& b)
   Calls XPRBloadbasis
void clearDir()
   Calls XPRBcleardir
int writeDir()
   Calls XPRBwritedir
int writeDir(const char *filename)
   Calls XPRBwritedir
int setSense(int dir)
   Calls XPRBsetsense
int getSense()
   Calls XPRBgetsense
char *getName()
   Calls XPRBgetprobname
int exportProb(int format, const char *filename)
   Calls XPRBexportprob
   Throws IOException (Java only)
int loadMat()
   Calls XPRBloadmat
int print()
   Calls XPRBprintprob  (only from C++)
int solve(const char *alg)
   Calls XPRBsolve
int minim(const char *alg)
   Calls XPRBminim
int maxim(const char *alg)
   Calls XPRBmaxim
int getProbStat()
   Calls XPRBgetprobstat
int getLPStat()
   Calls XPRBgetlpstat
int getMIPStat()
   Calls XPRBgetmipstat
int sync(int synctype)
   Calls XPRBsinc
double getObjVal()
   Calls XPRBgetobjval

int getNumIIS()
   Calls XPRBgetnumiis

void getIIS(ArrayList arrvar, ArrayList arrctr, int num)  (Java only)
   Calls XPRBgetiis

XPRBvar getVarByName(const char *name)
   Calls XPRBgetbyname

XPRBctr getCtrByName(const char *name)
   Calls XPRBgetbyname

XPRBsos getSosByName(const char *name)
   Calls XPRBgetbyname

XPRBindexSet getIISetByName(const char *name)
   Calls XPRBgetbyname

XPRBcut newCut(int id)
   Calls XPRBnewcut

XPRBcut newCut(XPRBlinRel& ac)
   Calls XPRBnewcut

XPRBcut newCut(XPRBlinRel& ac, int id)
   Calls XPRBnewcut

XPRBcut newCut()
   Calls XPRBnewcut

void delCut(XPRBcut& cut)
   Calls XPRBdelcut

int setCutMode(int mode)
   Calls XPRBsetcutmode

int addCuts(XPRBcut *cuts, int num)
   Calls XPRBaddcuts
XPRBvar

Description
Methods for modifying and accessing variables.

Constructors
- XPRBvar()
- XPRBvar(xbvar *v)

Methods
- xbvar *getCRef() (C++ only)
- bool isValid()
- int print()
  Calls XPRBprintvar
- int setType(int type)
  Calls XPRBsetvartype
- int setUB(double val)
  Calls XPRBsetub
- int setLB(double val)
  Calls XPRBsetlb
- int setLim(double val)
  Calls XPRBsetlim
- int fix(double val)
  Calls XPRBfixvar
- int setDir(int type, double val)
  Calls XPRBsetvardir
- int setDir(int type)
  Calls XPRBsetvardir
- const char *getName()
  Calls XPRBgetvarname
- int getColNum()
  Calls XPRBgetcolnum
- int getType()
  Calls XPRBgetvartype
- double getLB()
  Calls XPRBgetbounds
- double getUB()
  Calls XPRBgetbounds
- int getBounds(double *lb, double *ub) (C++ only)
  Calls XPRBgetbounds
- int getLim(double *val) (C++ only)
  Calls XPRBgetlim
- double getLim() (Java only)
  Calls XPRBgetlim
- double getSol()
  Calls XPRBgetsol
- double getRCost()
  Calls XPRBgetrcost
- double getRNG(int rngtype)
  Calls XPRBgetvarrng
XPRBctr

Description
Methods for modifying and accessing constraints and operators for constructing them.

Constructors
XPRBctr()
XPRBctr(xbctr *c)
XPRBctr(xbctr *c, XPRBlinRel& ctr)

Methods
xbctr *getCRef()   (C++ only)
bool isValid()
int print()
   Calls XPRBprintctr
int setType(int type)
   Calls XPRBsetctrtype
int setRange(double low, double up)
   Calls XPRBsetrange
int setModCut(bool mstat)
   Calls XPRBsetmodcut
bool isModCut()
   Calls XPRBgetmodcut
const char *getName()
   Calls XPRBgetctrname
double getRangeL()
   Calls XPRBgetrange
double getRangeU()
   Calls XPRBgetrange
int getRange(double *lw, double *up)   (C++ only)
   Calls XPRBgetrange
double getRHS()
   Calls XPRBgetrhs
int getRowNum()
   Calls XPRBgetrownum
int getType()
   Calls XPRBgetctrtype
double getSlack()
   Calls XPRBgetslack
double getAct()
   Calls XPRBgetact
double getDual()
   Calls XPRBgetdual
double getRNG(int rngtype)
   Calls XPRBgetctrrng
int setTerm(XPRBvar& var, double val)
   Calls XPRBsetterm
int setTerm(double val, XPRBvar& var)
   Calls XPRBsetterm
int setTerm(double val)
   Calls XPRBsetterm
int addTerm(XPRBvar& var, double val)
   Calls XPRBaddterm
int addTerm(double val, XPRBvar& var)
   Calls XPRBaddterm
int addTerm(XPRBvar& var)
   Calls XPRBaddterm
int addTerm(double val)
   Calls XPRBaddterm
int delTerm(XPRBvar& var)
   Calls XPRBdelterm
void add(XPRBlinExp& l)

Operators  (C++ only)
Assigning constraints and adding linear expressions:
   ctr = linrel
   ctr += linexp
   ctr -= linexp
XPRBcut

Description
Methods for modifying and accessing cuts and operators for constructing them.

Constructors
XPRBcut()
XPRBcut(xbcut *c)
XPRBcut(xbcut *c, XPRBlinRel& cut)

Methods
xbcut *getCRef()  (C++ only)
bool isValid()
int print()
  Calls XPRBprintcut
int setType(int type)
  Calls XPRBsetcuttype
int setID(int id)
  Calls XPRBsetcutid
double getRHS()
  Calls XPRBgetcutrhs
int getType()
  Calls XPRBgetcuttype
int getID()
  Calls XPRBgetcutid
int setTerm(XPRBvar& var, double val)
  Calls XPRBsetcutterm
int setTerm(double val, XPRBvar& var)
  Calls XPRBsetcutterm
int setTerm(double val)
  Calls XPRBsetcutterm
int addTerm(XPRBvar& var, double val)
  Calls XPRBaddcutterm
int addTerm(double val, XPRBvar& var)
  Calls XPRBaddcutterm
int addTerm(double val, XPRBvar& var)
  Calls XPRBaddcutterm
int addTerm(XPRBvar& var)
  Calls XPRBaddcutterm
int addTerm(double val)
  Calls XPRBaddcutterm
int delTerm(XPRBvar& var)
  Calls XPRBdelcutterm
void add(XPRBlinExp& l)

Operators  (C++ only)
Assigning cuts and adding linear expressions:
cut = linrel
cut += linexp
cut -= linexp
XPRBsos

Description
Methods for modifying and accessing Special Ordered Sets and operators for constructing them.

Constructors
XPRBsos()
XPRBsos(xbsos *s)
XPRBsos(xbsos *s, XPRBlinExp& l)

Methods
xbsos *getCRef() (C++ only)
bool isValid()
int print()
    Calls XPRBprintsos
const char *getName()
    Calls XPRBgetsosname
int getType()
    Calls XPRBgetsostype
int setDir(int type, double val)
    Calls XPRBsetsosdir
int setDir(int type)
    Calls XPRBsetsosdir
int addElement(XPRBvar& var, double val)
    Calls XPRBaddsosel
int addElement(double val, XPRBvar& var)
    Calls XPRBaddsosel
int delElement(XPRBvar& var)
    Calls XPRBdelsosel

Operators (C++ only)
Assigning and adding linear expressions to Special Ordered Sets:
set = linexp
set += linexp
XPRBIndexSet

Description
Methods for accessing index sets and operators for adding and retrieving set elements.

Constructors
XPRBIndexSet()
XPRBIndexSet(xbidxset *iset)

Methods
xbidxset *getCRef()  (C++ only)
bool isValid()
int print()
    Calls XPRBprintidxset
const char *getName()
    Calls XPRBgetidxsetname
int getSize()
    Calls XPRBgetidxsetsize
int addElement(const char *text)
    Calls XPRBaddidxel
int getIndex(const char *text)
    Calls XPRBgetidxel
const char *getIndexName(int i)
    Calls XPRBgetidxelname

Operators  (C++ only)
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]
XPRBbasis

Description
Methods for accessing bases.

Constructors
XPRBbasis()
XPRBbasis(xbasis *bs)

Methods
xbasis *getCRef() (C++ only)
bool isValid()
5.3.2 Additional classes

The classes listed in this section do not correspond to standard BCL modeling objects. Class XPRB contains some methods relating to the initialization and the general status of the software and in Java also the definition of all parameters (not listed here). This means, any parameter with the prefix XPRB_ in standard BCL has to be referred to as a constant of the Java class XPRB. For example, XPRB_BV in standard BCL becomes XPRB.BV in Java; in C++ it stays the same as in C.

Most of the remaining classes have been introduced to aid the termwise definition of constraints, by overloading certain arithmetic operators in the case of C++ and by overloading a set of simple methods in the case of Java. Linear expressions (class XPRBlinExp) are required in the definition of constraints and Special Ordered Sets. Quadratic expressions (class XPRBquadExp) are used to define quadratic objective functions. Linear relations (class XPRBlinRel), may be used as an intermediary in the definition of constraints.

In BCL Java, there are a few additional classes related to error handling and licensing, namely XPRBError, XPRBlicense, and XPRBlicenseError. 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 documentation of class XPRBError is listed below. XPRBlicenseError overloads this class without any methods of its own and is therefore not listed separately. The class XPRBlicense only serves for OEM licensing; for further detail please see the Xpress-MP OEM licensing documentation.

As in the previous section, wherever the C++ and Java functions are similar (remember that the type char or char * in C++ is replaced by String in Java, and the type bool in C++ becomes boolean in Java, there are no pointers in Java, and the return types more often are void, errors being dealt with by raising exceptions) they are listed only once, giving preference to the C++ notation.
**XPRBLinExp**

**Description**
Methods and operators for constructing linear expressions.

**Constructors**
- XPRBLinExp(double d)
- XPRBLinExp(int i)
- XPRBLinExp(double d, XPRBvar& v)
- XPRBLinExp(XPRBvar& v)
- XPRBLinExp(XPRBLinExp& l)

**Methods**
- double getSol()
- XPRBLinExp& neg()
- XPRBLinExp neg(XPRBLinExp& l)
- XPRBLinExp add(XPRBLinExp& l)
- XPRBLinExp add(XPRBvar& v)
- XPRBLinExp add(double d)
- XPRBLinExp add(XPRBLinExp& l1, XPRBLinExp& l2)
- XPRBLinExp add(XPRBvar& v, XPRBLinExp& l)
- XPRBLinExp add(double d, XPRBLinExp& l)
- int setTerm(XPRBvar& var, double val)
- int setTerm(double val, XPRBvar& var)
- int delTerm(XPRBvar& var)
- XPRBLinExp& mul(double d)
- XPRBLinExp mul(XPRBLinExp& l, double d)
- XPRBLinExp mul(double d, XPRBLinExp& l)
- XPRBLinExp& assign(XPRBLinExp& l)

**Java only**
- XPRBquadExp mul(XPRBvar& v)
- XPRBquadExp sqr()
- XPRBLinRel lEql(XPRBLinExp l)
- XPRBLinRel lEql(XPRBvar v)
- XPRBLinRel lEql(double d)
- XPRBLinRel gEql(XPRBLinExp l)
- XPRBLinRel gEql(XPRBvar v)
- XPRBLinRel gEql(double d)
- XPRBLinRel eql(XPRBLinExp l)
- XPRBLinRel eql(XPRBvar v)
- XPRBLinRel eql(double d)

**Operators**  
(C++ only)
Assigning (elements to) linear expressions:
- linexp1 += linexp2
- linexp1 -= linexp2
- linexp1 = linexp2
Composing linear expressions from linear expressions ($linexp$), variables ($var$) and double values ($val$). The following operators, with the exception of the negation of a linear expression, are defined outside any class definition:

- $linexp$
- $linexp1 + linexp2$
- $linexp1 - linexp2$
- $linexp * val$
- $val * linexp$
- $var * val$
- $val * var$
- $- var$
XPRBquadExp  (extends XPRBlinExp)

Description
Methods and operators for constructing quadratic expressions.

Constructors
XPRBquadExp()
XPRBquadExp(double d, XPRBvar& v1, XPRBvar& v2)
XPRBquadExp(XPRBvar& v)
XPRBquadExp(XPRBlinExp& l)
XPRBquadExp(XPRBquadExp& q)

Methods
XPRBquadExp& neg()
XPRBquadExp neg(XPRBquadExp q)
XPRBquadExp& add(XPRBquadExp& q)
XPRBquadExp& add(XPRBlinExp& l)
XPRBquadExp& add(XPRBvar& v)
XPRBquadExp& add(double d)
XPRBquadExp add(XPRBquadExp& q1, XPRBquadExp& q2)
int setTerm(XPRBvar& var1, XPRBvar& var2, double val)
int setTerm(double val, XPRBvar& var1, XPRBvar& var2)
int setTerm(XPRBvar& var, double val)
int setTerm(double val)
int delTerm(XPRBvar& var1, XPRBvar& var2)
int delTerm(XPRBvar& var)
XPRBquadExp& mul(double d)
XPRBquadExp mul(XPRBquadExp& q, double d)
XPRBquadExp mul(double d, XPRBquadExp& q)
XPRBquadExp& mul(XPRBquadExp& q)

Throws ArithmeticException ‘Non-quadratic expression’ if the result of the operation
is not quadratic

XPRBquadExp mul(XPRBquadExp& q1, XPRBquadExp& q2)

Throws ArithmeticException ‘Non-quadratic expression’ if the result of the operation
is not quadratic

XPRBquadExp mul(XPRBlinExp& l)

Throws ArithmeticException ‘Non-quadratic expression’ if the result of the operation
is not quadratic

XPRBquadExp mul(XPRBquadExp& q, XPRBlinExp& l)

Throws ArithmeticException ‘Non-quadratic expression’ if the result of the operation
is not quadratic

XPRBquadExp mul(XPRBlinExp& l, XPRBquadExp& q)

Throws ArithmeticException ‘Non-quadratic expression’ if the result of the operation
is not quadratic

XPRBquadExp mul(XPRBlinExp& l1, XPRBlinExp& l2)

Throws ArithmeticException ‘Non-quadratic expression’ if the result of the operation
is not quadratic

XPRBquadExp& assign(XPRBquadExp& q)

Operators  (C++ only)
Assigning (elements to) quadratic expressions:
qexp1 += qexp2
Composing quadratic expressions from quadratic and linear expressions (\( qexp \) and \( linexp \)), variables (\( var \)) and double values (\( val \)). The following operators, with the exception of the negation of a quadratic expression, are defined outside any class definition:

- \( qexp \)
- \( qexp1 + qexp2 \)
- \( qexp1 - qexp2 \)
- \( var1 \times var2 \)
- \( linexp \times var \)
- \( var \times linexp \)
- \( qexp \times val \)
- \( val \times qexp \)
- \( qexp1 \times qexp2 \)

Throws the exception ‘Non-quadratic expression’ if the result of the operation is not quadratic

C++ only

Functions outside any class definition that generate quadratic expressions:

- \( XPRBquadExp \) \( \text{sqr}(XPRBquadExp& \ Q) \)
- \( XPRBquadExp \) \( \text{sqr}(XPRBlinExp& \ l) \)
- \( XPRBquadExp \) \( \text{sqr}(XPRBvar& \ v) \)
XPRBlinRel  (extends XPRBlinExp)

Description
Methods and operators for constructing linear relations from linear expressions.

Constructors
XPRBlinRel(const XPRBlinExp& l, char t)
XPRBlinRel(const XPRBlinExp& l)
XPRBlinRel(const XPRBvar& v)

Operators  (C++ only)
Creating linear relations by establishing relations between linear expressions. The following operators are defined outside any class definition:
linexp1 <= linexp2
linexp1 >= linexp2
linexp1 == linexp2
XPRB

Description
Initialization and general settings (C++ and Java), definition of all parameters (Java only).

Constructors
XPRB() (Java only)
Throws XPRBlicenseError

Methods
int init() (C++ only)
Calls XPRBinit
XPRBlicense license() (Java only)
Calls XPRBinit
XPRBlicense license(String path)
int license(XPRBlicense lic)
void setOutputStream(Object stream) (Java only)
void setOutputStream()
int setColOrder(int num)
Calls XPRBsetcolorder
int setMsgLevel(int lev)
Calls XPRBsetmsglevel
int setRealFmt(String fmt)
Calls XPRBsetrealfmt
char *getVersion()
Calls XPRBgetversion
int getTime()
Calls XPRBgettime
**XPRBerror**  (extends *Error*)

**Description**
Exception raised by BCL errors (Java only).

**Methods**

- `String getMessage()`
- `int getErrorCode()`
Appendix
Appendix A

BCL error messages

There are two types of error messages displayed by BCL. Those marked ‘E’ (for Error) in the following list stop the execution of the program. Those marked ‘W’ (for Warning) do not interrupt the program. The marker ‘fct’ indicates that the name of the function where the error occurred will be printed out.

E-1502  *Not enough memory.*

   It is not possible to allocate the required amount of memory needed for BCL objects.

E-1504  *Dictionary cannot be re-initialized.*

   Dictionary sizes can only be set immediately after the creation of a problem.

E-1505  *(fct) No variable given.*

   Function *fct* requires a variable of type XPRBvar as an input parameter. Check whether the variable has been created (functions XPRBnewvar or XPRBnewarrvar).

E-1506  *(fct) No array of variables given.*

   Function *fct* requires an array of variables of type XPRBarrvar as an input parameter. Check whether the array has been created (function XPRBnewarrvar or alternatively functions XPRBstartarrvar and XPRBendarrvar).

E-1507  *(fct) No constraint given.*

   Function *fct* requires a constraint of type XPRBctr as an input parameter. Check whether the constraint has been created (functions XPRBnewctr, XPRBnewsum, XPRBnewarrsum, or XPRBnewprec).

E-1508  *(fct) No SOS given.*

   Function *fct* requires a SOS of type XPRBsos as an input parameter. Check whether the set has been created (functions XPRBnewtos, XPRBnewtosrc, or XPRBnewtosw).

E-1509  *(fct) No cut given.*

   Function *fct* requires a cut of type XPRBcut as an input parameter. Check whether the cut has been created (functions XPRBnewcut, XPRBnewcutsum, XPRBnewcutarrsum, or XPRBnewcutprec).

E-1510  *(fct) No basis given.*

   Function *fct* requires a basis of type XPRBbasis as an input parameter. Check whether the basis has been saved (function XPRBsavebasis).

E-1512  *(fct) No array of constants given.*

   Function *fct* requires an array of constants as an input parameter.

W-1513  *(fct) No variable given.*

   Function *fct* requires a variable of type XPRBvar as an input parameter. The command is being ignored.

W-1514  *(fct) No constraint given.*

   Function *fct* requires a constraint of type XPRBctr as an input parameter. The command is being ignored.
E-1515  *(fct) Problem has no ‘name’.*
The problem definition is incomplete (at least one variable and one constraint or one non-zero objective coefficient must be defined).

W-1516  *(fct) Problem has no ‘name’.*
The problem definition may be incomplete (at least one variable and one constraint or one non-zero objective coefficient must be defined).

W-1518  *(fct) No SOS given.*
Function \texttt{fct} requires a Special Ordered Set of type \texttt{XPRBsos} as an input parameter. The command is being ignored.

W-1519  *(fct) No cut given.*
Function \texttt{fct} requires a cut of type \texttt{XPRBcut} as an input parameter. The command is being ignored.

W-1520  *(fct) No solution available or problem modified since last solved.*
Function \texttt{fct} is trying to access solution information which is not available for the current problem.

E-1521  \texttt{Xpress-Optimizer error getting objective function value}.
The objective function value cannot be obtained from \texttt{Xpress-Optimizer}.

E-1522  \texttt{Xpress-Optimizer error getting ‘name’ status}.
\texttt{Xpress-Optimizer} solution status information cannot be obtained.

E-1523  \texttt{Unknown solution option ‘char’}.
Possible options for \texttt{XPRBsolve} include ‘\texttt{b},’ ‘\texttt{d},’ ‘\texttt{g},’ ‘\texttt{l},’ ‘\texttt{p}’. Refer to the reference manual for details.

E-1524  *(fct) Xpress-Optimizer error num during ‘name’.*
An \texttt{Xpress-Optimizer} error has occurred while executing the Optimizer function \texttt{name}. Refer to the Optimizer Reference Manual for details on the error number \texttt{num}.

W-1525  *(fct) Different problem loaded in Xpress-Optimizer.*
(Solution) information is being sought from the \texttt{Xpress-Optimizer} on a problem that is not the active problem in \texttt{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 \texttt{Xpress-Optimizer}. It may be necessary to solve the problem to access this information, or at least, load the matrix into \texttt{Xpress-Optimizer}.

E-1530  *(fct) Inconsistent bounds for variable ‘name’ (bdl,bdu).*
The lower bound is greater than the upper bound for the given variable.

E-1531  *(fct) Incorrect type for variable ‘name’.*
No type, or an incorrect type, has been specified for a variable. See the list of possible values in the reference manual (function \texttt{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 \texttt{XPRBsetctrtype}).

E-1536  *(fct) Inconsistent range for constraint ‘name’ (bdl,bdu).*
The lower range bound is greater than the upper bound for the given constraint.

E-1540  *(fct) Trying to modify a closed array of variables (‘name’).*
It is not possible to make changes to an array of variables after its definition has been terminated with \texttt{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 XPRBget_sostype).

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.

E-1560  No Xpress-BCL license found. Please contact Dash to obtain a license
        No valid BCL license has been found. If you did install a license, check whether you have copied it to the right place and that all environment variables and paths for BCL and the Xpress-Optimizer are set correctly.

E-1561  (fct) Initialization failed (value: num).
        Xpress-Optimizer could not be initialized (error code num).

W-1562  (fct) Working with Student License.
        BCL is running in Student mode; this mode implies restrictions to the available functionality and to the accepted problem size.

E-1563  (fct) Inconsistency during matrix generation.
        Internal error during the matrix generation.

E-1565  (fct) Internal error.
        Internal error during the matrix generation.

E-1566  Name too long: 'name'.
        A user-defined or BCL composed name exceeds the maximum length. (Remember that BCL adds indices to names if they already exist.)

E-1567  (fct) Size limits of the Student License exceeded.
        The specified model is too large to be run with the Student License.

W-1568  Operation fct not allowed in Student License.
        You are not authorized to execute function fct with the Student License of BCL.
W-1570  *XOSL:* text
Refer to the Optimizer Reference Manual for the indicated error.

E-1571  text
The initialization has not found a valid license.

W-1580  *Unknown output file format.*
Refer to the documentation of function XPRBexportprob for admissible output format options.

E-1582  *Internal error writing MPS file.*
Please contact the Dash support.

W-1587  *Switch to cut mode.*
The cut mode probably needs to be enabled (function XPRBsetcutmode) before this function is called.

E-1591  *(fct) Non-quadratic term.*
A term of the objective function has a power higher than 2.
Appendix B
Using BCL with the Optimizer library

BCL provides both modeling and basic optimization functions, which correspond to the functionality of Xpress-Mosel, or of the Xpress-Modeler and the functions of the Xpress-Optimizer library in ‘Console Mode’, respectively. However, if the user wishes to access the more advanced features of the Optimizer, obtain additional information, or change algorithm settings, the relevant Optimizer library functions have to be used directly.

The following sections explain in more detail how to use Optimizer library functions within a BCL program. The first section lists those functions which are compatible with BCL. It is followed by some general remarks about initialization, loading the matrix and the use of indices. The last section contains some typical examples for the use of BCL-compatible Optimizer functions in BCL programs.

Important: If a program uses Optimizer library functions the Optimizer header file has to be included in addition to the BCL header file. That is, the first lines of the program should contain the following:

```c
#include "xprb.h"
#include "xprs.h"
```

B.1 Switching between libraries

Generally speaking, there are two types of Optimizer library functions: those that access information about a problem or change settings for the search algorithms, and those that make changes to the problem definition. The first group of functions may be used in a BCL program without any problem. The second group requires the user to switch completely to the Optimizer library, for instance after a problem has been defined in BCL and the matrix has been loaded into the Optimizer.

B.1.1 BCL-compatible Optimizer functions

The following Optimizer library functions may be used with BCL (however, some caution is required with all functions that take column or row indices as input parameters, see Section B.4 below. Furthermore, the solution information in BCL is only updated automatically at the end of the search, in the global callbacks it needs to be updated by calling XPRBsync with the parameter XPRB_XPRS_SOL):

- **setting and accessing problem and control parameters**: functions XPRSsetintcontrol, XPRSgetintcontrol, XPRSgetintattrib etc.;
- **output and saving**: functions XPRSSave, XPRSSwritebasis, XPRSSrange, XPRSSis, XPRSSwriteptsol, XPRSSwritesol, XPRSSwriteprtrange, XPRSSwriterange, XPRSSgetlpsol, XPRSSgetmipsol, XPRSSwriteomni, XPRSSwriteprob, 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, XPRSloloaddirs, XPRSloloadbasis, XPRSloloadbasis, XPRSloloadsecurevecs, XPRStscale, XPRStftran, XPRStbtran, **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 XPRSad..., XPRSchg..., XPRSdel...;

• **solution algorithms**: XPRSminim, XPRSmaxim, XPRSGlobal;

• **input of data or problem(s)**: XPRStreadprob, XPRSloloadlp, XPRSloloadglobal, XPRSloloadglobal, XPRSloloadqdp, XPRSalter, XPRStsetprobname;

• **manipulation of the matrix**: XPRStrestore;

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, XPRStinit, from a user program. In standard use of BCL the function XPRBnewprob calls the BCL initialization function XPRBinit that automatically initializes the Optimizer if this is the first call to XPRBinit. In very large applications or integration with other systems it may be preferable to call XPRBinit explicitly to separate the initialization from the definition of the problem(s).

At the end of the program, the normal BCL termination routine should be applied, first releasing any memory associated to problems using XPRBdelprob and subsequently calling XPRBfree to tidy up. These routines also free memory associated with the Optimizer library and hence neither of the XPRStdestroyprob or XPRStfree 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:

```c
XPRBprob prob;
prob = XPRBnewprob("Example1");
... /* Define and solve the problem */
```

Integration of a BCL problem into some larger application:

```c
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:

```c
XPRBarrvar x;
...
x = XPRBnewarrvar(prob, 10, XPRB_PL, "x", 0, 100);
... /* (Define the rest of the problem) */
XPRBmaxim(prob,""); /* Load matrix and maximize LP problem */
for(i=0; i<10; i++) /* Print solution values for variables */
  printf("x%g: %d, ",XPRBgetvarname(prob,x[i]),
        XPRBgetsol(prob,x[i]));
```

Switch to using the Optimizer library after problem input with BCL:

```c
XPRBprob bcl_prob;
XPRSprob opt_prob;
XPRBarrvar x;
int i, cols, len;
double sol;
char names;

bcl_prob = XPRBnewprob("Example1"); /* Initialize BCL (and the Optimizer library) and create a new problem */
x = XPRBnewarrvar(bcl_prob, 10, XPRB_PL, "x", 0, 100);
... /* Define the rest of the problem */
XPRBloadmat(bcl_prob); /* Load matrix into the Optimizer */
opt_prob = XPRBgetXPRSprob(bcl_prob); /* Get the Optimizer problem */
XPRSmaxim(opt_prob,"" ); /* Maximize the LP problem */
XPRSgetintattrib(opt_prob,XPRS_COLS, &cols); /* Get the number of columns */
sol = malloc(cols * sizeof(double));
XPRSgetlpsol(opt_prob,sol,NULL,NULL,NULL); /* Get entire primal solution */
XPRSgetintattrib(opt_prob,XPRS_NAMELENGTH,&len); /* Get the length of names (divided by 8) */
names = malloc((8*len+1)*cols); /* Get variable names */
for(i=0; i<cols; i++) /* Print all solution values */
  printf("x%g: %g, ", names+((8*len+1)*i), sol[i]);
```

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 (depending on the setting of the control parameter XPRS_SOLUTIONFILE).

B.5 Using BCL-compatible functions

The Optimizer library functions that are most likely to be used in a BCL program are those for setting and accessing control and problem parameters, as shown in the following examples. The control parameters can be set and accessed at any time after the software has been initialized (see Section B.2). The problem attributes only return the problem-specific values once the problem has been loaded into the Optimizer. Note that all the parameters take their default values at the beginning of a BCL program but they are not reset if several problems are solved in a single program and changes are made to the parameter values along the way.

Setting control parameters:

```csharp
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 */
XPRBmaxim(bcl_prob,""); /* Load matrix and maximize LP problem */
```

Accessing problem parameters:

```csharp
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_ROWS, &rows); /* Get number of rows */
XPRBmaxim(bcl_prob,""); /* Maximize the LP problem */
```

Another likely set of functions are the Optimizer library callbacks for solution printout and possibly for directing the branch and bound search (see the remarks about indices in Section B.4):

```csharp
void XPRS_CC_printsol(XPRSprob opt_prob, void *my_object)
{
    XPRBprob bcl_prob
    XPRBvar x;
    int num;

    bcl_prob = (XPRBprob)my_object;
```
**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:**

```c++
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_MIPADDCCUTOFF, 0.999); // Set an ADDCUTOFF value
XPRSgetintattrib(opt_prob, XPRS_ROWS, &rows); // Get number of rows
bcl_prob.maxim("MIP"); // Maximize the LP problem
```

**Using Xpress-Optimizer callbacks:**

```c++
void XPRS_CC_printsol(XPRSprob opt_prob, void *my_object) {
    XPRBprob *bcl_prob
    XPRBvar x;
    int num;
    
    bcl_prob = (XPRBprob*)my_object;
    XPRSgetintattrib(opt_prob, XPRS_MIPSOLS, &num); // Get number of the solution
    XPRSsetintcontrol(opt_prob, XPRS_SOLUTIONFILE, 0);
    bcl_prob->sync(XPRB_XPRS_SOL); // Update BCL solution values
    cout << "Solution " << num << ": Objective value: ";
    cout << bprob->getObjVal() << endl;
}
```
x = bcl_prob->getVarByName("x_1");
if(x.getColNum()>-1) // Test whether variable is in the matrix
    cout << x.getName() << ": " << x.getSol() << endl;
XPRSsetintcontrol(opt_prob, XPRS_SOLUTIONFILE, 1);
}

int main(int argc, char **argv)
{
    XPRBprob bcl_prob;
    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.maxim("g"); // Maximize the MIP problem
}

As in the C case, it is possible within a BCL program written in C++ to switch entirely to Xpress-Optimizer (see Section B.3).

B.7 Using the Optimizer with BCL Java

Starting with Release 3.0 of BCL it is possible to combine BCL Java problem definition with direct access to the Optimizer problem in Java. All that is said in the previous sections about BCL-compatible functions remains true. The only noticeable difference is that the Optimizer Java needs to be initialized explicitly (by calling XPRSinit) before the Optimizer problem is accessed.

The following are Java implementations of the code extracts showing the use of BCL-compatible functions:

Setting and accessing parameters (this code throws the exceptions 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.ROWS);
    /* Get number of rows */
    bcl_prob.maxim("""); /* Maximize the LP problem */

Using Xpress-Optimizer callbacks:

    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 {
    num = oprob.getIntAttrib(XPRS.MIPSOLS);
    /* Get number of the solution */
    oprob.setIntControl(XPRS.SOLUTIONFILE, 0);
    bcl_prob.sync(XPRB.XPRS_SOL);
    /* Update BCL solution values */
    System.out.println("Solution "+ num + ": Objective value: " +
        bcl_prob.getObjVal());
    x = bcl_prob.getVarByName("x_1");
    if(x.getColNum()>=1) /* Test whether variable is in the matrix */
        System.out.println(x.getName() + ": " + x.getSol());
    oprob.setIntControl(XPRS.SOLUTIONFILE, 1);
} catch(XPRBprobException e) {
    System.out.println("Error "+ e.getCode() + ": " + e.getMessage());
}
}

public static void main(String[] args) throws XPRSException {
    XPRB bcl;
    XPRBprob bcl_prob;
    XPRSprob opt_prob;
    IntSolCallback cb;
    XPRBVar x;
    bcl = new XPRB(); /* Initialize BCL */
    bcl_prob = bcl.newProb("Example1"); /* Create a new problem in BCL */
    XPRS.init(); /* Initialize Xpress-Optimizer */

    x = bcl_prob.newVar("x_1", XPRB_BV); /* Define a variable */
    /* Define the rest of the problem */
    opt_prob = bcl_prob.getXPRSprob();
    cb = new IntSolCallback();
    opt_prob.addIntSolListener(cb, bcl_prob);
    /* Define an integer solution callback */
    bcl_prob.maxim("g"); /* Maximize the MIP problem */
}
Appendix C

Working with cuts in BCL

This chapter describes an extension to BCL that enables the user to define cuts in a similar way to constraints. Although cuts are just additional constraints, they are treated differently by BCL. To start with, they are defined as a separate type (XPRBcut instead of XPRBct.r). Besides the type, the following differences between the representation and use of constraints and cuts in BCL may be observed:

- Cuts cannot be non-binding or ranged.
- Cuts are not stored with the problem, this is up to the user.
- Cuts have no names, but they have got an integer indicating their classification or identification number.
- Function XPRBdelcut deletes the cut definition in BCL, but does not influence the problem in Xpress-Optimizer if the cut has already been added to it.
- Cuts are added to the problem while it is being solved without having to regenerate the matrix; they can only be added to the matrix (using function XPRBaddcuts) in one of the callback functions of the Xpress-Optimizer cut manager (see the Xpress-Optimizer manual). Furthermore, they can only be defined on variables that are already contained in the matrix.

The following functions are available in BCL for handling cuts:

- XPRBaddcutarrterm: Add multiple terms to a cut.  
  p. 28
- XPRBaddcuts: Add cuts to a problem.  
  p. 26
- XPRBaddcutterm: Add a term to a cut.  
  p. 27
- XPRBdelcut: Delete a cut definition.  
  p. 42
- XPRBdelcutterm: Delete a term from a cut.  
  p. 43
- XPRBgetcutid: Get the classification or identification number of a cut.  
  p. 62
- XPRBgetcutrhs: Get the RHS value of a cut.  
  p. 63
- XPRBgetcuttype: Get the type of a cut.  
  p. 64
- XPRBnewcut: Create a new cut.  
  p. 103
- XPRBnewcutarrsum: Create a sum cut with individual coefficients (i ci xi).  
  p. 104
- XPRBnewcutprec: Create a precedence cut (v1+dur v2).  
  p. 105
- XPRBnewcutsum: Create a sum cut (i xi).  
  p. 106
- XPRBprintcut: Print out a cut.  
  p. 118
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.

```c
#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;
    XPRSprob bprob;

    bprob = (XPRSprob)vd; /* Get the BCL problem */
    XPRSgetintattrib(oprob, XPRS_NODES, &num);
    if(num == 2) /* Only generate cuts at node 2 */
    {
        /* ca0: s_1+2 <= s_0 */
        ca[0] = XPRBnewcutprec(bprob, start[1], 2, start[0], 2);
        ca[1] = XPRBnewcut(bprob, XPRB_L, 2); /* ca1: 4*s_2 - 5.3*s_3 <= -17 */
        XPRBaddcutterm(ca[1], start[2], 4);
        XPRBaddcutterm(ca[1], start[3], -5.3);
        XPRBaddcutterm(ca[1], NULL, -17);
        printf("Adding constraints:
    ");
        for(i=0;i<2;i++) XPRBprintcut(ca[i]);
        if(XPRBaddcuts(bprob, ca, 2)) printf("Problem with adding cuts.
    ");
        return 0; /* Call this func. once per node */
    }

    return(0);
}

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

    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 */
    XPRBsetcbcutmgr(oprob, usrcme, prob); /* Def. the cut manager callback */
    XPRBSolve(prob, "g"); /* Solve the MIP problem */
    ... /* Solution output */
    return 0;
}
```

C.1 Example

Working with cuts in BCL 199 BCL Reference Manual
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.

```cpp
#include <iostream>
#include "xprb_cpp.h"
#include "xprs.h"
using namespace std;
using namespace ::dashoptimization;
...
XPRBvar start[NJ];
XPRBprob p("Jobs"); // Initialize BCL and a new problem
int XPRS_CC usrcme(XPRSprob oprob, void* vd)
{
  XPRBcut ca[2];
  int num;
  int i=0;
  XPRBprob *bprob;
  bprob = (XPRBprob*)vd; // Get the BCL problem
  XPRSgetintattrib(oprob, XPRS_NODES, &num);
  if(num == 2) // Only generate cuts at node 2
  {
    ca[0] = bprob->newCut(start[1]+2 <= start[0], 2);
    cout << "Adding constraints:" << endl;
    for(i=0;i<2;i++) ca[i].print();
    if(bprob->addCuts(ca,2)) cout << "Problem with adding cuts." << endl;
  }
  return 0; // Call this function once per node
}
int main(int argc, char **argv)
{
  XPRSprob oprob;
  for(j=0;j<4;j++) start[j] = p.newVar("start");
  ... // Define constraints and an objective function
  oprob = p.getXPRSprob(); // Get Optimizer problem
  p.setCutMode(1); // Enable the cut mode
  XPRSsetcbcutmgr(oprob, usrcme, &p); // Def. the cut manager callback
  p.solve("g"); // Solve the problem as MIP
  ... // Solution output
  return 0;
}
```

C.3  Java version of the example

As is explained in Section B.7, before accessing directly the problem held in Xpress-Optimizer we need to initialize explicitly the Optimizer Java. The cut manager callback is implemented in Java by the class ‘cutMgrListener’.

```java
import java.io.*;
import com.dashoptimization. *;

public class xcbutex
{
  ...
  static XPRBVar[] start;
  static XPRB bcl;

  static class CutMgrCallback implements XPRScutMgrListener
  {
    public int XPRScutMgrEvent(XPRSprob oprob, Object data)
    {
      // Handle cut event...
    }
  }
```

XPRBprob bprob;
XPRBcut[] ca;
int num,i;

bprob = (XPRBprob)data; /* Get the BCL problem */

try {
    num = oprob.getIntAttrib(XPRS.NODES);
    if(num == 2) /* Only generate cuts at node 2 */ {
        ca = new XPRBcut[2];
        ca[0] = bprob.newCut(start[1].add(2) .lEql(start[0]), 2);
        ca[1] = bprob.newCut(start[2].mul(4) .add(start[3].mul(-5.3)) .lEq(-17), 2);
        System.out.println("Adding constraints:");
        for(i=0;i<2;i++) ca[i].print();
        bprob.addCuts(ca);
    }
    catch(XPRSprobException e) {
        System.out.println("Error "+ e.getCode() + ": "+ e.getMessage());
    }
    return 0; /* Call this method once per node */
}

public static void main(String[] args) throws XPRSexception {
    XPRBprob p;
    XPRSprob oprob;
    CutMgrCallback cb;

    bcl = new XPRB(); /* Initialize BCL */
    p = bcl.newProb("Jobs"); /* Create a new problem */
    XPRS.init(); /* Initialize Xpress-Optimizer */

    start = new XPRBvar[4]; /* Create 'start' variables */
    for(j=0;j<4;j++) start[j] = p.newVar("start");
    /* Define constraints and an objective function */

    oprob = p.getXPRSprob(); /* Get Optimizer problem */
    p.setCutMode(1); /* Enable the cut mode */
    cb = new CutMgrCallback();
    oprob.addCutMgrListener(cb, p); /* Def. the cut manager callback */
    p.solve("g"); /* Solve the problem as MIP */
    /* Solution output */
}
## Symbols

- $\times$, 180, 182
- $\div$, 180, 182
- $\pm$, 173–176, 179, 181
- $\Rightarrow$, 180, 182
- $\rightarrow$, 173, 174, 179, 182
- $\leq$, 183
- $\Rightarrow$, 173–175, 179, 182
- $\rightarrow$, 183
- $\Rightarrow$, 183
- $\left[\right]$, 176

## A

- activity value, 9, 53
- add
  - index element, 29
- add, 173, 174, 179, 181
- addElement, 175, 176
- addTerm, 172, 174
- array, 11, 154
  - add entry, 128
  - append, 34
  - create, 6
  - delete, 39
  - incremental definition, 6
  - name, 7, 54
  - print, 18
  - size, 7, 55
  - terminate, 49
- assign, 179, 181

## B

- basis
  - class, 177
  - delete, 10, 40
  - load, 10, 96
  - save, 10, 127
- BCL
  - version number, 6, 93
- bound
  - fix, 6, 52
  - get, 6, 56
  - integer limit, 6, 71, 139
  - lower, 6, 138
  - semi-continuous limit, 6, 71, 139
  - upper, 6, 149
- branching directive
  - SOS, 17, 147
  - variable, 6, 150

## C

- callback
  - error messages, 5, 20, 37
  - messages, 5, 20, 38

### printing, 5, 20, 38

### change
  - constraint type, 8
  - variable type, 6, 152
- clearDir, 169
- column order, 129
- constraint
  - activity, 9, 53
  - add array, 7, 25
  - add term, 7, 33
  - add terms, 25
  - change type, 8, 130
  - class, 172
  - creation, 7, 102
  - definition, 7, 102
  - delete, 8, 41
  - delete coefficient, 8
  - delete term, 48
  - dual, 9, 65
  - get range, 8, 79
  - get RHS, 8, 81
  - get type, 8, 61
  - index, 8, 82
  - model cut, 8, 74
  - name, 8, 59
  - number, 82
  - precedence, 109
  - print, 18, 117
  - ranging information, 60
  - set coefficient, 7
  - set model cut, 8, 140
  - set range, 8, 144
  - set term, 148
  - set type, 130
  - slack, 9, 84
  - sum, 114
  - sum with coefficients, 100
- create
  - index set, 107, 159, 163
- cut
  - add array, 28
  - add term, 27
  - add terms, 28
  - class, 174
  - classification, 62
  - creation, 103
  - definition, 103
  - delete, 42
  - delete term, 43
  - get RHS, 63
  - get type, 64
  - identification, 62
  - model, 8, 74, 140
  - precedence, 105
print, 118
set classification, 131
set identification, 131
set term, 133
set type, 134
sum, 106
sum with coefficients, 104
switch mode, 132
cut mode, 132
cuts, 26
D
data
input, 124, 125, 159, 163
reading, 124, 125
data input, 14
decimal sign
change, 5, 135
delCtr, 168
delElement, 175
delete
array, 39
constraint, 8, 41
constraint coefficient, 8
constraint term, 48
cut, 42
problem, 4, 44
quadratic terms, 45
set element, 17
set member, 47
SOS, 17, 46
delQObj, 168
delSos, 169
delTerm, 173, 174, 179, 181
dictionary
size, 136
directive
delete, 35
SOS, 17, 147, 159, 163
variable, 6, 13, 150
directives
file, 156
dual values, 9, 65
E
E-1502, 187
E-1504, 187
E-1505, 187
E-1506, 187
E-1507, 187
E-1508, 187
E-1509, 187
E-1510, 187
E-1512, 187
E-1515, 188
E-1521, 188
E-1522, 188
E-1523, 188
E-1524, 188
E-1526, 188
E-1530, 188
E-1531, 188
E-1535, 188
E-1536, 188
E-1540, 188
E-1541, 189
E-1542, 189
E-1543, 189
E-1545, 189
E-1546, 189
E-1547, 189
E-1550, 189
E-1560, 189
E-1561, 189
E-1563, 189
E-1565, 189
E-1566, 189
E-1567, 189
E-1571, 190
E-1582, 190
E-1591, 190
eql, 179
error
exit, 5, 137
error callback, 5, 20, 37
error handling, 5, 20, 137
error message, 5, 20, 37
exportProb, 169
expression
linear, 178
quadratic, 178
F
file
reading, 14
find by name, 57
fix, 171
format
real numbers, 145
G
garbage collection, 165
generate matrix, 97
gEq1, 179
getAct, 172
getBounds, 171
getColNum, 171
getCRef, 167, 168, 171, 172, 174–177
getCtrByName, 167, 170
getDual, 172
getErrorCode, 185
getID, 174
getIIS, 170
getIndex, 176
getIndexName, 176
getIndexSetByName, 170
getLB, 171
getLimit, 171
getLPStat, 169
getMessage, 185
getMIPStat, 169
getName, 169, 171, 172, 175, 176
getNumIIS, 170
getObjVal, 170
getProbStat, 169
getRange, 172
getRangeL, 172
I
IIS, see irreducible infeasible set
incremental definition
array, 6
index
  constraint, 8, 82
  variable, 6, 58
index set, 14
  add element, 14, 29
  class, 176
  create, 14, 159, 163
  creation, 107
  element name, 14, 67
  find element, 14, 66
  index number, 14, 66
  name, 14, 68
  print, 18, 120
  size, 14, 69
init, 184
initialization, 4, 95, 110, 189, 190
input
  decimal sign, 5, 135
  file, 124, 125, 159, 163
input file, 14
interface pointer, 89, 151
  callbacks, 36
irreducible infeasible set
  constraints, 70
  number, 75
  variables, 70
isModCut, 172
isValid, 167, 171, 172, 174–177
L
lEql, 179
licence, 4
license, 95, 189, 190
license, 184
linear expression
  class, 179
linear relation, 178
  class, 183
load matrix, 5
loadBasis, 169
loadMat, 169
M
matrix
add cuts, 26
column ordering, 129
generation, 97
loading, 5
output, 18, 50, 159, 163
matrix generation
  column order, 129
maxim, 169
maximize, 9, 98, 153
message level, 5, 141, 165
minim, 169
minimize, 9, 99, 153
MIPRESOLVE, 193
model cut, 8, 140
modeling object
  finding, 57
mul, 179, 181
N
name
  array, 7, 54
  composing of, 108
  constraint, 8, 59
dictionary, 136
  index set, 14, 68
  index set element, 14, 67
  problem, 77
  SOS, 17, 86
  variable, 6, 90
namespace, 157
neg, 179, 181
newCtr, 168
newInstance, 169
newSos, 169
newVar, 168
number
  IIS, 75
O
objective
  add quadratic term, 30
delete quadratic terms, 45
get sense, 8, 83
  quadratic, 19
  set quadratic term, 143
  set sense, 8, 146
  value, 9, 76
objective function, 142
optimize, 9, 153
output
  file, 18, 50
  redirection, 165
output level, 5, 141, 165
output stream, 165
P
package, 161
partial integer
  get limit, 6, 71
  set limit, 6, 139
precedence constraint, 8, 109
precedence cut, 105
PRESOLVE, 193
print
array, 18, 116
constraint, 18, 117
cut, 118
index set, 18, 120
problem, 18, 121, 159, 163
program output, 119
SOS, 18, 122
text, 119
variable, 18, 123
print, 169, 171, 172, 174-176
print flag, 5, 141
printing
decimal sign, 5, 135
printing callback, 5, 20, 38
priority, 13
delete, 35
SOS, 17, 147
variable, 6, 150
problem
add cuts, 26
class, 168
delete, 4, 44
delete basis, 10, 40
file output, 18, 50
initialization, 4, 110
load basis, 10, 96
LP status, 10, 72
maximize, 98
minimize, 99
MIP status, 10, 73
objective value, 9, 76
output, 4, 121
print, 18, 159, 163
reset, 126
save basis, 10, 127
solve, 9, 153
status, 10, 78
program output
print, 18, 119
Q
QP, 19
quadratic expression
class, 181
Quadratic Programming, 19
quadratic term, 30, 143
quadratic terms
delete, 45
R
range, 8, 79, 144
get values, 8
ranging information
constraint, 60
variable, 91
read
data line, 124, 125
reduced cost value, 9, 80
reference constraint, 17, 112
reset
problem, 126
reset, 168
RHS, 8, 63, 81
running time, 88
S
saveBasis, 169
scheduling, 10
security system, 4
semi-continuous
get limit, 6, 71
set limit, 6, 139
semi-continuous integer
get limit, 6, 71
set limit, 6, 139
sense
objective function, 8, 83, 146
set
cut mode, 132
index, 14
setColOrder, 168, 184
setDictionarySize, 168
setDir, 171, 175
setID, 174
setLB, 171
setLim, 171
setModCut, 172
setMagLevel, 168, 184
setObj, 168
setOutputStream, 168, 184
setRange, 172
setRealPnt, 168, 184
setSense, 169
setTerm, 172, 174, 179, 181
setType, 171, 172, 174
setUB, 171
size
array, 7, 55
dictionary, 136
index set, 14, 69
size limit, 189
slack values, 9, 84
solution, 9, 85
objective, 9, 76
solve, 9, 98, 99, 153
solve, 169
SOS
add array, 17, 31
add element, 17
add member, 32
add members, 31
class, 175
creation, 16, 111-113, 159, 162
delete, 17, 46
delete element, 17
delete member, 47
name, 17, 86
print, 18, 122
set directive, 17, 147
type, 17, 87
weights, 113
sqr, 179, 182
status
LP, 10, 72
MIP, 10, 73
problem, 10, 78
Student Edition, 118
Student mode, 189
student version, 20, 50, 117, 120-122
sum constraint, 8, 100, 114
sum cut, 106
sync, 169

T
table
  sparse, 14
time, 88
type
  constraint, 8, 61, 130
cut, 64, 134
SOS, 17, 87
variable, 6, 92, 152

V
variable
  array, 154
  array of, 6, 101
  change type, 6, 152
  class, 171
  creation, 6, 115
deletion callback, 36
  fix value, 6, 52
get bounds, 6, 56
get limit, 6, 71
get type, 6, 92
index, 6, 58
interface pointer, 89, 151
lower bound, 6, 138
name, 6, 90
number, 58
print, 18, 123
print array, 116
ranging information, 91
reduced cost, 9, 80
set directive, 6, 150
set limit, 6, 139
set type, 152
solution, 9, 85
upper bound, 6, 149
variable types, 1
version number, 6, 93

W
W-1513, 187
W-1514, 187
W-1516, 188
W-1518, 188
W-1519, 188
W-1520, 188
W-1525, 188
W-1551, 189
W-1552, 189
W-1562, 189
W-1568, 189
W-1570, 190
W-1580, 190
W-1587, 190
writeDir, 169

X
xbcut, 103
XPRB, 184
XPRB_ARR, 57
XPRB_BV, 92, 101, 115, 152, 178
XPRB_CTR, 57
XPRB_DICT_IDX, 136
XPRB_DICT_NAMES, 136
XPRB_DIR, 78
XPRB_DN, 147, 150
XPRB_E, 61, 64, 100, 102–104, 106, 114, 130, 134
XPRB_ERR, 37
XPRB_FGETS, 124, 125
XPRB_G, 61, 64, 100, 102–104, 106, 114, 130, 134
XPRB_GEN, 78
XPRB_IDX, 57
XPRB_L, 61, 64, 100, 102–104, 106, 114, 130, 134
XPRB_LCOST, 91
XPRB_LOACT, 60, 91
XPRB_LP, 50
XPRB_LP_CUTOFF, 72
XPRB_LP_CUTOFF_IN_DUAL, 72
XPRB_LP_INFEAS, 72
XPRB_LP_OPTIMAL, 72
XPRB_LP_UNBOUNDED, 72
XPRB_LP_UNFINISHED, 72
XPRB_LP_UNSOLVED, 72
XPRB_MAXIM, 83, 146
XPRB_MINIM, 83, 146
XPRB_MIP_INFEAS, 73
XPRB_MIP_LP_NOT_OPTIMAL, 73
XPRB_MIP_LP_OPTIMAL, 73
XPRB_MIP_NO_SOL_FOUND, 73
XPRB_MIP_NOT_LOADED, 73
XPRB_MIP_OPTIMAL, 73
XPRB_MIP_SOLUTION, 73
XPRB_MOD, 78
XPRB_MPS, 50
XPRB_N, 61, 100, 102, 114, 130
XPRB_PD, 147, 150
XPRB_PI, 92, 101, 115, 152
XPRB.PL, 92, 101, 115, 152
XPRB.PR, 147, 150
XPRB_PU, 147, 150
XPRB_R, 61
XPRB_SI, 87, 111–113
XPRB_S2, 87, 111–113
XPRB_SC, 92, 101, 115, 152
XPRB_SI, 92, 101, 115, 152
XPRB_SOL, 78
XPRB_SOS, 57
XPRB_UCOST, 91
XPRB_UDN, 60, 91
XPRB_UI, 92, 101, 115, 152
XPRB_UP, 147, 150
XPRB_UPACT, 60, 91
XPRB_VAR, 57
XPRB_WAR, 37
XPRB_XPRS_PROB, 155
XPRB_XPRS_SOL, 155
XPRBaddarterm, 25
XPRBaddcutarterm, 28
XPRBaddcuts, 26, 42
XPRBaddcutterm, 27
XPRBadddixel, 29
XPRBaddqterm, 20, 30
XPRBaddsosarrel, 31