
AIMMS Language Reference - AIMMS Programming Interface

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com.

Copyright © 1993–2018 by AIMMS B.V. All rights reserved.

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

Email: info@aimms.com
WWW: www.aimms.com

AIMMS is a registered trademark of AIMMS B.V. IBM ILOG CPLEX and CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Artelys. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation. \TeX , \LaTeX , and $\AMS-\LaTeX$ are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of AIMMS B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from AIMMS B.V.

AIMMS B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall AIMMS B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, AIMMS B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, AIMMS B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.

This documentation was typeset by AIMMS B.V. using \LaTeX and the LUCIDA font family.

Chapter 34

The AIMMS Programming Interface

In addition to the capability to call external procedures and functions from within an AIMMS application, AIMMS also provides a generic Application Programming Interface (API). This chapter describes the semantics of the complete AIMMS API, and provides an extended example to familiarize you with its use. In addition, it discusses the concurrency aspects when multiple external applications are controlling a single AIMMS session. Note that this chapter assumes that you have some basic knowledge of the C programming language.

This chapter

34.1 Introduction

One can think of several scenario's in which a path of communication needs to be set up between AIMMS and an external software component. The two most common scenario's are listed below.

Communication scenario's

- You have a collection of functions within an external DLL which you want to use to perform certain data manipulations within your AIMMS model through calls to an `ExternalProcedure` or `ExternalFunction`.
- From within your own application you want to open an AIMMS project, pass input data to it, solve an optimization model, and retrieve the solution.

The most straightforward method to set up communication between AIMMS and an external DLL is by calling an external procedure or function from within your model. If, during such a call, the data of one or more scalar or low-dimensional indexed identifiers need to be passed to the DLL, the easiest way to exchange this data is by passing either a single scalar value or a (dense) array of scalar values as arguments to the corresponding DLL function. For higher-dimensional identifiers, however, the memory requirements for passing array arguments may grow out of hand, and additional control may be needed.

Exchanging data

With only the possibility to call external procedures and functions from within an AIMMS model, however, you have no possibility, from within an external application, to

Application-controlled execution

- open an AIMMS project,

- initiate the exchange of data, or
- execute one or more procedures in your model.

The AIMMS Application Programming Interface (API) described in this chapter addresses both the drawbacks associated with dense data transfer, and the need to control the execution of an AIMMS model from within an external application. Figure 34.1 provides a schematic overview of the capabilities to communicate using both the concept of external procedures and functions and the AIMMS API. Left-to-right arrows are implemented through external proce-

The AIMMS API

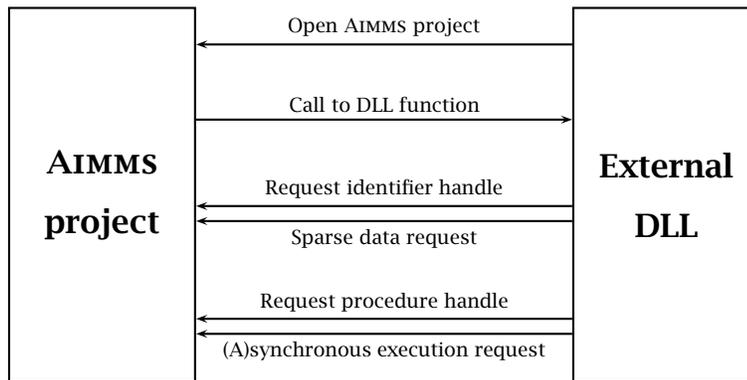


Figure 34.1: Interaction between AIMMS and an external DLL

cedure and function calls within your model, while all right-to-left arrows are provided for by the AIMMS API.

Central to the AIMMS API is the concept of *handles*. Handles are represented by unique integer numbers, and provide indirect access to named identifiers and procedures within an AIMMS model. Access to the associated objects within the model is through the functions of the API. With every identifier or procedure in the model, multiple handles can be associated, each of which may behave differently when passed to a function in the AIMMS API depending on its declaration or on the sequency of API functions previously applied to it (e.g. during sparse data retrieval). Handles can be created by AIMMS and passed as arguments to a DLL function, or can be created from within an external application.

Handles

Through the functions in the AIMMS API, you can initiate further actions on a given identifier or procedure handle from within an external application. More specifically, the API functions allow you to

API functions

- obtain information about identifiers in the model, such as domain, range and type,

- set up sparse data communication between an identifier in the AIMMS model and an external application, and
- request either synchronous or asynchronous execution of a procedure within the AIMMS model.

AIMMS only provides a C interface to the functions in its API. When you are using a different language which requires a different interface, you should implement the required interface yourself in C/C++ or in a compatible language.

C interface

This remainder of this section will provide you with a simple ExternalProcedure declaration and the associated C function that illustrates the basic use of the AIMMS API and further familiarizes you with the basic concepts. Because of the many API functions and their interdependence, it is practically impossible to provide illustrative examples for each API function separately in the context of the this language reference. Therefore, the subsequent sections will only explain the semantics of each separate API function.

Single example

The following C function accepts the name of an AIMMS identifier with double-valued values. It queries AIMMS for a handle to that identifier, the corresponding domain and all associated values. For the sake of conciseness, the DLL function does not check all return values passed by the AIMMS API functions.

*Example:
printing
identifier info*

```
#include <stdio.h>
#include <string.h>
#include <aimmsapi.h>

DLL_EXPORT(void) print_double_aimms_identifier_info(char *name) {
    int handle, full, sliced, domain[AIMMSAPI_MAX_DIMENSION],
        tuple[AIMMSAPI_MAX_DIMENSION], storage, i;
    char file[256], buffer[256];
    FILE *f;

    AimmsValue value;
    AimmsString strvalue;

    /* Create a handle associated with the identifier name passed */
    AimmsIdentifierHandleCreate(name, NULL, NULL, 0, &handle);

    /* Get the dimension, domain and storage type of the identifier
       associated with the handle */
    AimmsAttributeDimension (handle, &full, &sliced);
    AimmsAttributeRootDomain(handle, domain);
    AimmsAttributeStorage (handle, &storage);

    if ( storage != AIMMSAPI_STORAGE_DOUBLE ) return;

    /* Open a file consisting of the identifier name with the extension .def,
       and print the identifier's name and dimension */

    strcpy(file, name); strcat(file, ".def");
    if ( ! (f = fopen(file, "w")) ) return;
    fprintf(f, "Identifier name: %s\n", name);
}
```

```

fprintf(f, "Dimension      : %d\n", full);

/* Prepare strvalue to hold the locally declared buffer */
strvalue.String = buffer;

/* Print a header containing the names of the domain sets */
fprintf(f, "\nData values   : \n");
for ( i = 0; i < full; i++ ) {
    strvalue.Length = 256;
    AimmsAttributeName(domain[i], &strvalue); fprintf(f, "%17s", buffer);
}
fprintf(f, "%16s\n", "Double value");
for ( i = 0; i < full; i++ ) fprintf(f, "%17s", "-----");
fprintf(f, "\n");

/* Print all tuples with nondefault data values */
AimmsValueResetHandle(handle);
while ( AimmsValueNext(handle, tuple, &value) ) {
    for ( i = 0; i < full; i++ ) {
        strvalue.Length = 256;
        AimmsSetElementToName(domain[i], tuple[i], &strvalue);
        fprintf(f, "%17s", buffer);
    }
    fprintf(f, "%17.5f\n", value.Double);
}

fclose(f);
}

```

If the DLL function is part of a DLL "Userfunc.dll", then it can be called from within AIMMS by the following ExternalProcedure declaration.

```

ExternalProcedure PrintParameterInfo {
    Arguments : (param);
    DLLName   : "Userfunc.dll";
    BodyCall  : print_double_aimms_identifier_info(string scalar: param);
}

```

Its only argument is an element parameter into the predefined set AllIdentifiers. It can therefore be called with any identifier name.

```

ElementParameter param {
    Range      : AllIdentifiers;
    Property   : input;
}

```

Consider a two-dimensional parameter TransportCost(i,j) which contains the following data. *Call example*

```

TransportCost := DATA TABLE
                Rotterdam      Antwerp      Berlin
!             -----
Amsterdam     1.00            2.50            10.00
Rotterdam     1.20            10.00
Antwerp       11.00
;

```

Then the procedure call `PrintParameterInfo('TransportCost')` will result in the creation of a file `TransportCost.def` with the following contents.

```
Identifier name: TransportCost
Dimension      : 2

Data values   :
Cities        Cities        Double value
-----
Amsterdam     Rotterdam      1.00000
Amsterdam     Antwerp        2.50000
Amsterdam     Berlin         10.00000
Rotterdam     Antwerp        1.20000
Rotterdam     Berlin         10.00000
Antwerp       Berlin         11.00000
```

The prototypes of all the available AIMMS API functions, as well as all C macro definitions that are relevant for the execution of the API functions are provided in a single header file `aimmsapi.h`. You should include this header file in all your source files that make use of the AIMMS API functions.

*aimmsapi.h
header file*

The AIMMS API functions are provided in two flavors: ASCII and Unicode. For each of the functions mentioned in this chapter, there is an implementation postfixed with `A` for the ASCII flavor, and an implementation postfixed with `W` for the Unicode flavor. For instance, when `UNICODE` is defined, a call to `AimmsIdentifierHandleCreate` will be mapped to the implemented function `AimmsIdentifierHandleCreateW`. For the Unicode flavor, on Windows, double-byte character arrays are used to communicate strings corresponding to the UTF-16LE character encoding. For the Unicode flavor, on Linux, quadruple-byte character arrays are used to communicate strings corresponding to the UTF-32LE character encoding. This corresponds to the `wchar_t*` type on both platforms. Please make sure that the option `external_string_character_encoding` is set to corresponding encoding. For the ASCII flavor, both on Windows and on Linux, multibyte character arrays are used, and the encoding is determined by the option `external_string_character_encoding`.

*Two flavors of
API*

The AIMMS API functions are provided in the form of a Visual C/C++ import library `libaimms3.lib` to the `libaimms3.dll` DLL, which can be included in the link step of your external AIMMS DLL. When you are using the Visual C/C++ compiler, this import library will take care that all the relevant API functions are imported from the AIMMS executable when your AIMMS application loads the external DLL. For other compilers, you should consult the compiler documentation on how to import the functions in `libaimms3.dll` into your program.

*libaimms3.lib
import library*

All AIMMS API functions provide an integer return value. When the requested operation has succeeded, the value `AIMMSAPI_SUCCESS` is returned. When the operation has failed, AIMMS will return the value `AIMMSAPI_FAILURE`. In the latter case, you can obtain an error code and string through the API function `Aimms-APILastError` (see also Section 34.7).

Return values

AIMMS will only allow you to pass or create handles for identifier types with which data is associated, i.e. sets, parameters and variables. In addition, you can pass or create handles to suffixes of identifiers as long as the resulting suffix results in a set or parameter.

Only identifiers with data

34.2 Obtaining identifier attributes

For every identifier handle passed to (or created from within) an external function, AIMMS can provide additional attributes that are either related to the declaration of the identifier associated with the handle, or to the particular identifier slice that was passed as an argument in the external function call. Table 34.1 lists all AIMMS API functions which can be used to obtain these additional attributes.

Obtaining attributes

<code>int AimmsAttributeName(int handle, AimmsString *name)</code>
<code>int AimmsAttributeType(int handle, int *type)</code>
<code>int AimmsAttributeStorage(int handle, int *storage)</code>
<code>int AimmsAttributeDefault(int handle, AimmsValue *value)</code>
<code>int AimmsAttributeSetUnit(int handle, char *unit, char *convention)</code>
<code>int AimmsAttributeGetUnit(int handle, AimmsString *unitName)</code>
<code>int AimmsAttributeDimension(int handle, int *full, int *slice)</code>
<code>int AimmsAttributeRootDomain(int handle, int *domain)</code>
<code>int AimmsAttributeDeclarationDomain(int handle, int *domain)</code>
<code>int AimmsAttributeCallDomain(int handle, int *domain)</code>
<code>int AimmsAttributeRestriction(int handle, int *domainhandle)</code>
<code>int AimmsAttributeSlicing(int handle, int *slicing)</code>
<code>int AimmsAttributePermutation(int handle, int *permutation)</code>
<code>int AimmsAttributeFlagsSet(int handle, int flags)</code>
<code>int AimmsAttributeFlagsGet(int handle, int *flags)</code>
<code>int AimmsAttributeFlags(int handle, int *flags)</code>
<code>int AimmsAttributeElementRange(int handle, int *sethandle)</code>
<code>int AimmsAttributeCompoundDimension(int handle, int *dim)</code>
<code>int AimmsAttributeCompoundDomain(int handle, int *domain)</code>

Table 34.1: AIMMS API functions for obtaining handle attributes

With the functions `AimmsAttributeName` and `AimmsAttributeType` you can request the name and identifier type of the identifier associated with a handle. AIMMS passes the name of an identifier through an `AimmsString` structure (explained below). AIMMS only allows handles for identifier types with which data can be associated. More specifically, AIMMS distinguishes the following identifier types:

Identifier name and type

- simple root set,
- simple subset,
- relation (i.e. a compound set not used for indexing),
- compound root set,
- compound subset,
- indexed set,
- numeric parameter,
- element parameter,
- string parameter,
- unit parameter,
- variable,
- element variable.

When the handle refers to a suffix of an identifier, the suffix type is appended to the identifier name separated by a dot.

In addition to the identifier type, AIMMS also associates a storage type with each handle. It is the data type in which AIMMS expects the data values associated with the handle to be communicated. The function `AimmsAttributeStorage` returns the storage type. The possible storage types are:

Storage type

- double (double),
- integer (int),
- binary (int, but only assuming 0-1 values),
- string (char *).

The complete list of identifier and storage type values returned by these functions can be found in the header file `aimmsapi.h`.

With the function `AimmsAttributeDefault` you can obtain the default value of the identifier associated with a handle. The default value can either be a double, integer or string value, depending on the storage type associated with the handle. Below you will find the convention used by AIMMS to pass such storage type dependent values back and forth.

Default value

Through the functions `AimmsAttributeGetUnit` and `AimmsAttributeSetUnit` you can get and set the units of measurement (see also Chapter 32) that will be used when passing data from and to AIMMS. When setting the unit to be used, you can specify both a unit and a convention. AIMMS will parse the given unit expression and use the specified convention to compute the appropriate multiplication factors between the internal and external representation of the data of the identifier at hand.

Setting and getting units

All transfer of integer, double or string values takes place through the record structures `AimmsString` and `AimmsValue` defined as follows.

Passing integer, double or string values

```
typedef struct AimmsStringRecord {
    int Length;
    char *String;
} AimmsString;
```

```
typedef union AimmsValueRecord {
    double    Double;
    int       Int;
    struct {
        int    Length;
        char   *String;
    }
} AimmsValue;
```

When value is such a structure, you can obtain an integer, double or string value through value.Int, value.Double or value.String, respectively.

For strings, you must set value.Length to the length of the string buffer passed through value.String before calling the API function. When AIMMS fills the value.String buffer, the actual length of the string passed back is assigned to value.Length. When the actual string length exceeds the buffer size, AIMMS truncates the string passed back through value to the indicated buffer size, and assigns the length of the actual string to value.Length.

Passing string lengths

For each handle you can obtain the dimension of the associated identifier by calling the function AimmsAttributeDimension. The function returns:

Identifier dimensions

- the *full* dimension of the identifier as given in its declaration, and
- the *slice* dimension, i.e. the resulting dimension of the actual identifier slice associated with the handle.

AIMMS uses tuples of length equal to the full dimension whenever information is communicated regarding the index domain of a handle or its slicing. When explicit data values associated with a handle are passed using the AIMMS API functions discussed in Section 34.4, AIMMS communicates such values using tuples of length equal to the slice dimension.

For all data communication with external DLLs AIMMS considers sets to be represented by binary indicator parameters indexed over their respective (simple or compound) root sets. For all elements in these root sets, such an indicator parameter assumes the value 1 if a root set element (or tuple of root set elements) is contained in the set at hand, or 0 otherwise. Since the default of these indicator parameters is 0, AIMMS only needs to communicate the nonzero values, i.e. exactly the tuples that are actually contained in the set. In connection with this representation, AIMMS returns the following (full or slice) dimensions for sets:

Set dimensions

- the dimension of a *simple set* is 1,
- the dimension of a *relation* is the dimension of the Cartesian product of which the relation is a subset,
- the dimension of a *compound set* is 1,
- the dimension of an *indexed set* is the dimension of the index domain of the set plus 1.

The functions `AimmsAttributeRootDomain`, `AimmsAttributeDeclarationDomain` and `AimmsAttributeCallDomain` can be used to obtain an integer array containing handles to domain sets for every dimension of the identifier at hand. These domains play a different role in the sparse data communication, as explained below.

Identifier domains

The function `AimmsAttributeRootDomain` returns an array of handles to the respective root sets associated with the index domain specified in the identifier's declaration. You need these handles, for instance, to obtain a string representation of the element numbers returned by the data communication AIMMS API functions discussed in Section 34.4.

Root domain handles

The function `AimmsAttributeDeclarationDomain` returns an array of handles to the respective domain sets specified in the identifier's declaration. These domain sets can be equal to their corresponding root sets, or to subsets thereof. AIMMS will only pass data values for element tuples in the declaration domain, unless you have specified the raw translation modifier (see also Section 11.2) for a handle argument, or have created the handle yourself with the raw flag set (see also Section 34.3).

Declaration domain handles

The function `AimmsAttributeCallDomain` returns an array of handles to the particular subsets of the root sets (as returned in the root domain of the handle) to which data communication is restricted for this handle. The call domain can be different from the global domain if an actual external argument has been restricted to a subdomain of the root set in an external call (see also Section 10.3), or if you have created the handle with an explicit call domain yourself (see also Section 34.3). AIMMS will only pass data values associated with element tuples in just the call domain (raw flag set), or in the intersection of the call and declaration domain (raw flag not set).

Call domain handles

With the function `AimmsAttributeRestriction` you can obtain a handle to the global domain restriction of an indexed identifier as specified in its declaration and (dynamically) maintained by AIMMS as necessary. You may want to use this handle in conjunction with raw handles (explained in Section 34.4) to verify whether a particular element satisfies its domain restriction.

Domain restriction

Consider the following set and parameter declarations.

Example

```
Set S_0 {
  Index      : i_0;
}
Set S_1 {
  SubsetOf   : S_0;
  Index      : i_1, j_1;
}
```

```

Set S_2 {
  SubsetOf : S_1;
  Index    : i_2;
}
Parameter p {
  IndexDomain : i_0;
}
Parameter q {
  IndexDomain : (i_1, j_1) | p(i_1);
}

```

A handle to (in AIMMS notation) $q(i_1, i_2)$ will return handles to

- S_0 and S_0 for the respective root domains,
- S_1 and S_1 for the respective declaration domains,
- S_1 and S_2 for the respective call domains, and
- $p(i_1)$ for the domain restriction.

As discussed in Section 10.3, the actual arguments in a procedure or function call can be slices of higher-dimensional identifiers within your model. When the slice dimension of a handle in an external call is less than its full dimension, you can use the function `AimmsAttributeSlicing` to find out which dimensions of the associated AIMMS identifier have been sliced, and to which elements. The function returns an integer array containing, for every dimension, the element number (within the associated root set) to which the corresponding domain has been sliced, or the number `AIMMSAPI_NO_ELEMENT` if no slicing took place.

Slicing

Through the function `AimmsAttributePermutation` you can obtain the permutation of a permuted handle created with the function `AimmsAttributeHandleCreatePermuted`. The output permutation argument must be an integer array of length equal to the full dimension of the identifier. AIMMS returns the following values:

Domain permutations

- if a dimension of the handle is sliced, the corresponding position in the permutation array will be 0,
- if a dimension is not sliced, the corresponding position in the permutation array will contain the sliced position (starting at 1, and numbered from 1 to the handle's slice dimension)
 - in which AIMMS will store elements of the corresponding dimension in a tuple returned by the functions `AimmsValueNext` and `AimmsValueNextMulti`, or
 - in which AIMMS expects such elements in calls to the functions `AimmsValueSearch` and `AimmsValueRetrieve`.

By specifying the input-output type and the `ordered`, `retainspecials`, `elementsasordinals` or `raw` translation modifiers for arguments in an external call (see also Section 11.2), you can influence the manner in which data is passed to an external function. With the AIMMS API function `AimmsAttributeFlagsGet` you obtain the active set of flags indicating whether

Getting ordered, special, raw and read-only flags

- the data associated with a handle is passed ordered (ordered flag),
- special values are passed unchanged or are translated (`retainspecials` flag),
- element tuples are passed by their element numbers (`elementsasordinals` flag),
- inactive data is passed (raw flag), and
- you can make assignments to the handle (input-output type).

The result is the *bitwise or* function of the individual flag values as defined in the `aimmsapi.h` header file.

Through the function `AimmsAttributeFlagSet` you can modify the flag settings for an existing handle. Note that the result of calls to `AimmsValueNext` may become unpredictable after modifying the ordered flag. In such a case, you are advised to reset the handle through the function `AimmsHandleReset`.

Setting flags

When a handle is associated with an element parameter within your application, you can use the function `AimmsAttributeElementRange` to obtain a handle to the set constituting the element range of the element parameter. You need this handle, for instance, when you want to obtain a string representation of the element numbers within the element range communicated by AIMMS in the AIMMS API functions discussed Section 34.4.

Element range

As discussed above, AIMMS considers a compound set as a 1-dimensional domain for the communication with external DLLs. You can use the functions `AimmsAttributeCompoundDimension` and `AimmsAttributeCompoundDomain` to retrieve the dimension of the compound set as a relation, as well as handles to the associated global domain sets. You can use these to translate the compound element numbers into tuples of element names or ordinal numbers in the respective domain sets using the AIMMS API functions discussed in Section 34.5.

Compound dimension and domain

34.3 Managing identifier handles

AIMMS offers the capability to dynamically create and delete handles to any desired identifier slice over any desired local subdomain from within a DLL. In addition, a subset of the AIMMS data control operators (as discussed in Section 25.3) can be called from within external DLLs. Table 34.2 lists all available AIMMS API functions for creating handles and performing data control operations.

Creation and data control

```

int AimmsIdentifierHandleCreate(char *name, int *domain, int *slicing,
                               int flags, int *handle)
int AimmsIdentifierHandleCreatePermuted(char *name, int *domain, int *slicing,
                                         int *permutation, int flags, int *handle)
int AimmsIdentifierHandleDelete(int handle)
int AimmsIdentifierEmpty(int handle)
int AimmsIdentifierCleanup(int handle)
int AimmsIdentifierUpdate(int handle)
int AimmsIdentifierDataVersion(int handle, int *version)

```

Table 34.2: AIMMS API functions for handle management

You can use the function `AimmsIdentifierHandleCreate` to dynamically create a handle to (a slice of) an AIMMS identifier or a suffix thereof within an external function or procedure. You can restrict the scope of a handle by

Creating a handle

- specifying a *call* domain to which you want to restrict the handle, or
- by *slicing* one or more dimensions of the identifier.

If you want a handle to an identifier itself, the name passed to `AimmsIdentifierHandleCreate` should just be the identifier name. If you want a handle to a suffix of an identifier, you should pass the name of the identifier followed by a dot and the suffix name. Thus, for instance, you should pass the name "Transport.ReducedCost" if you want a handle to the reduced costs of the variable Transport.

Obtaining a suffix

When you want to create a handle over the full root domain, you can simply pass a null pointer for the `domain` argument. If you want to specify an additional call domain, you must pass an integer array of length equal to the identifier's full dimension, each element containing a handle to the set to which you want to restrict the domain. If the `raw` flag is not set, passing a null pointer for the domain handle will effectively restrict the declaration domain of the identifier at hand, because of the semantics of the `raw` flag (see also Sections 34.2 and 34.4).

Specifying a call domain

When you want to create a handle over the full dimension of an identifier, you can simply pass a null pointer for the `slicing` argument. If you want to create a handle to a slice, you must pass an integer array of length equal to the identifier's full dimension, each element containing either a null element for all the domains that you do not want to slice, or the element number of the element to which you want to slice.

Specifying a slice

With the `flags` argument in a call to `AimmsIdentifierHandleCreate` you can specify which modification flags should be set for the handle to be created. The format of the `flags` argument is the same as in the function `AimmsAttributeFlags` discussed in the previous section.

Modification flags

With the function `AimmsIdentifierHandleCreatePermuted` you can obtain a handle to a multidimensional identifier, for which the order in which element tuples are returned is permuted. Handles created by `AimmsIdentifierHandleCreatePermuted` are always read-only, i.e. cannot be used in the functions `AimmsValueAssign` and `AimmsValueAssignMulti`. The permutation argument must be specified according to the rules explained for the function `AimmsAttributePermutation`.

Creating a permuted handle

Consider an identifier $p(i, j, k, l)$ for which you want to retrieve the values as if the identifier were defined as $p(k, i, l, j)$. To retrieve all values of p in this order, the permutation array must be specified as `[2,4,1,3]`.

Example

With the function `AimmsIdentifierHandleDelete` you can delete a dynamically created handle that is no longer needed. The function fails when you try to delete a handle that was passed as an argument to the DLL. After deletion the handle can no longer be used in conjunction with any AIMMS API function.

Deleting handles

The AIMMS API functions

- `AimmsIdentifierEmpty`,
- `AimmsIdentifierCleanup`, and
- `AimmsIdentifierUpdate`

Empty, cleanup and update handles

can be called to perform the identical actions on a set or identifier (slice) from within an external DLL as can be accomplished by the data control operators `EMPTY`, `CLEANUP` and `UPDATE` from within AIMMS, respectively. The function `AimmsIdentifierEmpty` will empty the particular slice and subdomain of the identifier associated with the handle. The other two functions will cleanup or update the *entire* data set of the identifier associated with the handle, regardless of the specified slicing and local domain.

For every identifier within your model AIMMS maintains a version number of the data associated with the identifier. This number is incremented each time a data value of the identifier has been changed. You can use the function `AimmsIdentifierDataVersion` to retrieve this version number, for instance, to verify whether the data has changed relative to the last time you retrieved it.

Data version

When you apply the function `AimmsIdentifierDataVersion` to the predefined handle value `AIMMSAPI_MODEL_HANDLE`, AIMMS will return a data version number based on the cases and datasets currently active within the model. AIMMS will update this number as soon as the combined configuration of the active case and/or datasets within the model has changed, as well as after a call to the `CLEANDEPENDENTS` operator. A change in this global data version number is a good indication that the contents of all or a number of domain sets may have changed, and must be retrieved again.

Checking for global data changes

34.4 Communicating individual identifier values

With every identifier handle AIMMS lets you retrieve all associated nondefault data values on an element-by-element basis. In addition, AIMMS lets you search whether a nondefault value exists for a particular element tuple, and make assignments to individual element tuples. Table 34.3 lists all the available AIMMS API functions for this purpose.

Communicating identifier values

<code>int AimmsValueCard(int handle, int *card)</code>
<code>int AimmsValueResetHandle(int handle)</code>
<code>int AimmsValueSearch(int handle, int *tuple, AimmsValue *value)</code>
<code>int AimmsValueNext(int handle, int *tuple, AimmsValue *value)</code>
<code>int AimmsValueNextMulti(int handle, int *n, int *tuples, AimmsValue *values)</code>
<code>int AimmsValueRetrieve(int handle, int *tuple, AimmsValue *value)</code>
<code>int AimmsValueAssign(int handle, int *tuple, AimmsValue *value)</code>
<code>int AimmsValueAssignMulti(int handle, int n, int *tuples, AimmsValue *values)</code>
<code>int AimmsValueDoubleToMapval(double value, int *mapval)</code>
<code>int AimmsValueMapvalToDouble(int mapval, double *value)</code>

Table 34.3: AIMMS API functions for sparse data communication

The function `AimmsValueCard` returns the cardinality of a handle, i.e. the number of nondefault elements of the associated identifier slice. You can call this function, for instance, when you need to allocate memory for the data structures in your own code before actually retrieving the data.

Cardinality

The functions `AimmsValueResetHandle`, `AimmsValueSearch` and `AimmsValueNext` retrieve nondefault values associated with a handle on an element-by-element basis.

Retrieving nondefault values

- The function `AimmsValueResetHandle` resets the handle to the position just before the first nondefault element.
- The function `AimmsValueSearch` expects an input tuple of element numbers (in the slice domain), and returns the first tuple for which a nondefault value exists on or following the input tuple.
- The function `AimmsValueNext` returns the first nondefault element directly following the element returned by the last call to `AimmsValueNext` or `AimmsValueSearch`, or the first element if the function `AimmsValueResetHandle` was called last. The function fails when there is no such element.

By calling `AimmsValueResetHandle` and subsequently `AimmsValueNext` it is possible to retrieve *all* nondefault values. By calling the function `AimmsValueSearch` you can directly skip to a particular element tuple if you have found that the intermediate tuples are not interesting anymore, and continue from there.

The functions `AimmsValueResetHandle`, `AimmsValueNext` and `AimmsValueSearch` do not accept handles to scalar (i.e. 0-dimensional) identifier slices. To retrieve and assign scalar values you should use the functions `AimmsValueRetrieve` and `AimmsValueAssign` explained below.

No scalar handles

The particular element returned by the functions `AimmsValueSearch` and `AimmsValueNext` may differ depending on the setting of the `ordered` flag for the handle. If the handle has been created `unordered` (default), the values returned successively are ordered by increasing element number in a right-to-left tuple order. If the handle has been created `ordered`, AIMMS will return values in accordance with the ordering principles imposed on all *local* tuple domains.

Unordered versus ordered retrieval

By default, AIMMS will only pass values for element tuples that lie within the *current* contents of the intersection of the call domain and declaration domain of an identifier. Thus, the values that get passed may depend on a dynamically changing domain restriction that is part of the index domain in the declaration of an identifier. When the `raw` modification flag is set for a handle, AIMMS will pass *all* available data values in the call domain, regardless of the domain restrictions.

Raw data retrieval

All data retrieval functions return a tuple and the associated nondefault value. The interpretation of the `value` argument for all possible storage types was discussed on page 576. The `tuple` argument must be an integer array of length equal to the *slice* dimension of the handle. Upon success, the tuple contains the element numbers in the *global* domain sets for every non-sliced dimension.

Return tuple and value

By setting the flag `elementsasordinals` during the creation of a handle, you can modify the default tuple representation. If this flag is set, the tuples returned by AIMMS will contain ordinal numbers corresponding to the respective call domains associated with the handle. Similarly, AIMMS expects tuples that are passed to it, to contain ordinal numbers as well, when this flag is set.

Element or ordinal numbers

While at first sight the choice for representing tuples by their element numbers in the *global* domain of a handle may seem less convenient than ordinal numbers in its call domain, you must be aware that the latter representation is not invariant under changes in the contents of the call domain. Alternatively to setting the flag `elementsasordinals`, you can also convert the returned element numbers into these formats using the AIMMS API functions discussed in Section 34.5.

Rationale

The expected storage type of the data values returned by the data retrieve functions can be obtained using the function `AimmsAttributeStorage`. The possible storage types for the various identifier types are listed below:

Value types

- numeric parameters and variables return double or integer values,

- all set types return binary values,
- element parameters return integer element numbers, and
- string and unit parameters return string values.

The element numbers returned for element parameters are relative to the set handle returned by the function `AimmsAttributeElementRange`. You can use the AIMMS API functions of Section 34.5 to obtain the associated ordinal numbers or string representations.

*Element
parameter
values*

For sets (either simple, relation, compound or indexed), the data retrieval functions return the binary value 1 for just those elements (or element tuples) that are contained in the set. For compound sets the tuple argument contains the compound element number which can be converted into its corresponding tuple representation using the function `AimmsSetCompoundToTuple` (see also Section 34.5). For indexed sets, AIMMS returns tuples for which the last component is the (simple or compound) element number of an element contained in the set slice associated with all but the last tuple components.

Set values

When a handle to a numeric parameter or variable has been created with the special flag set, the data retrieval functions will pass any special number value associated with the handle as is (see also Sections 11.2 and 34.2). AIMMS represents special numbers as double precision floating point numbers outside AIMMS' ordinary range of computation. The function `AimmsValueDoubleToMapval` returns the `MapVal` value associated with any double value (see also Table 6.1), while the function `AimmsValueMapvalToDouble` returns the double representation associated with any type of special number.

*Converting
special numbers*

The function `AimmsValueRetrieve` returns the value for a specific element tuple in the slice domain. This value can be either the default value or a nondefault value. The tuple must consist of element numbers in the corresponding domain sets. When the raw flag is not set, the function fails (but still returns the default value of the associated identifier) for any tuple outside of the index domain of the handle. When the raw flag is set, the function fails only when there is no data for the tuple.

*Retrieving
specific values*

The function `AimmsValueAssign` lets you assign a new value to a particular element tuple in the slice domain. If you want to assign the default value you can either pass a null pointer for value, or a pointer to the appropriate default value. The function fails if you try to assign a value to an element tuple outside the contents of the call domain of the handle. When the raw flag is not set, the function will also fail if the assigned tuple lies outside of the current (active) contents of the declaration domain.

*Assigning
values*

When a particular identifier handle requires the exchange of a large amount of values, you are strongly encouraged to use the functions `AimmsValueNextMulti` and `AimmsValueAssignMulti` instead of the functions `AimmsValueNext` and `AimmsValueAssign`. In general, AIMMS can perform the simultaneous exchange of multiple values much more efficient than the equivalent sequence of single exchanges. For both functions, the `tuples` array must be an integer array of length `n` times the *slice* dimension of the handle, while the `values` array must be the corresponding `AimmsValue` array of length `n`.

Exchanging multiple values

- In the function `AimmsValueNextMulti`, AIMMS will fill the `tuples` array with the respective tuples for which nondefault values are returned in the `values` array. Upon return, the `n` argument will contain the actual number of values passed.
- In the function `AimmsValueAssignMulti`, the `tuples` array must be filled sequentially with the respective tuples to which the assignments take place via the `values` array.

When your data transfer involves the addition of a large amount of set elements to an AIMMS set as well, you may also want to consider using the function `AimmsSetAddElementMulti` (see Section 34.5).

When a handle corresponds to a 0-dimensional (i.e. scalar) identifier slice, you can still use the `AimmsValueRetrieve` and `AimmsValueAssign` to retrieve its value or assign a value to it. In this case, the `tuple` argument is ignored.

Communicating scalar values

When you want to delete or add an existing element or element tuple to a set, you must assign the value 0 or 1 to the associated tuple respectively. If you want to add a tuple of nonexisting simple elements, you must first add these elements to the corresponding global simple domain sets using the function `AimmsSetAddElement` discussed below. Similarly, if you want to add nonexisting compound elements to a compound subset, you must first add such elements to the corresponding compound root set using the function `AimmsSetAddTupleToCompound`.

Assigning set values

34.5 Accessing sets and set elements

The AIMMS API functions discussed in the previous section allow you to retrieve and assign individual values of (slices of) indexed identifiers associated with tuples of set element numbers used by AIMMS internally. The AIMMS API functions discussed in this section allow you to add elements to simple and compound sets, and let you convert element numbers into ordinal numbers and element names, or vice versa. Table 34.4 presents all set related API functions.

Accessing sets

<pre>int AimmsSetAddElement(int handle, char *name, int *element) int AimmsSetAddElementMulti(int handle, int n, int *elementNumbers) int AimmsSetAddElementRecursive(int handle, char *name, int *element) int AimmsSetRenameElement(int handle, int element, char *name) int AimmsSetDeleteElement(int handle, int element)</pre>
<pre>int AimmsSetElementNumber(int handle, char *name, int allowCreate, int *elementNumber, int *isCreated) int AimmsSetAddElementMulti(int handle, int n, int *elementNumbers) int AimmsSetAddElementRecursiveMulti(int handle, int n, int *elementNumbers)</pre>
<pre>int AimmsSetElementToOrdinal(int handle, int element, int *ordinal) int AimmsSetElementToName(int handle, int element, AimmsString *name) int AimmsSetOrdinalToElement(int handle, int ordinal, int *element) int AimmsSetOrdinalToName(int handle, int ordinal, AimmsString *name) int AimmsSetNameToElement(int handle, char *name, int *element) int AimmsSetNameToOrdinal(int handle, char *name, int *ordinal)</pre>
<pre>int AimmsSetCompoundToTuple(int handle, int compound, int *tuple) int AimmsSetTupleToCompound(int handle, int *tuple, int *compound) int AimmsSetAddTupleToCompound(int handle, int *tuple, int *compound) int AimmsSetAddTupleToCompoundRecursive(int handle, int *tuple, int *compound)</pre>

Table 34.4: AIMMS API functions for passing set data

The function `AimmsSetAddElement` allows you to add new element names to a simple set. AIMMS will return with the internal element number assigned to the element, which you can use for further references to the element. The function fails if an element with the specified name already exists, but still sets element to the corresponding element number. To add a new element tuple to a compound root set, you should use the function `AimmsSetAddTupleToCompound` discussed below.

Adding elements to simple sets

If the set is a subset, AIMMS will add the element to that subset only. Thus, the function will fail and return no element number if the corresponding element does not already exist in the associated root set. If the element is present in the root set, but not in the domain of the subset, the functions will fail but still return the element number corresponding to the presented string. With the function `AimmsSetAddElementRecursive` you can add an element to a subset itself as well as to all its supersets, up to the associated root set.

Adding element to subsets

Through the function `AimmsSetRenameElement` you can provide a new name for an element number associated with an existing element in a set. The change in name does not imply any change in the data previously defined over the element. However, the element will be displayed according to its new name in the graphical user interface, or in data exchange with external data sources.

Renaming set elements

With the function `AimmsSetDeleteElement` you can delete the element with the given element number from either a simple or compound set. If the set is a simple or compound *root* set, any remaining data defined over the element in subsets parameters and variables will become inactive. To remove such

Deleting set elements

inactive references to the deleted element, you can use the API function `AimmsIdentifierCleanup` (see also Section 34.3).

Alternatively to applying the functions `AimmsSetAddElement` and `AimmsSetDeleteElement` to subsets, you can also use the function `AimmsValueAssign` to modify the contents of a subset. In that case, you should assign the value 1 to the tuple that should be added to the subset, or 0 to a tuple that should be removed (as discussed in the previous section). The function `AimmsValueAssign` will also work on indexed sets and relations.

Modifying subset contents

When, as part of a large data transfer from an external data source to AIMMS, you have to add a large amount of (non-existing) set elements to a simple AIMMS set, the use of the functions `AimmsSetElement` and `AimmsSetElementRecursive` may become a performance bottleneck compared to any bulk data transfer of multidimensional data defined over these set elements. The reason for this is that the function `AimmsSetElementAdd` adds elements one at a time, and may need to extend the internal data structures used to store set data many times, which is a relatively expensive action.

Adding multiple set elements to a simple set...

As an alternative, AIMMS offers a different set of functions that combined allow you to add multiple set elements much more efficiently, at the expense of a slightly more complex sequence of actions. The functions are:

... in an efficient manner

- the function `AimmsSetElementNumber`, which retrieves an existing, or creates a new, set element number for a given element name, *but does not add it yet to any set*, and
- the functions `AimmsSetAddElementMulti` and `AimmsSetAddElementRecursiveMulti`, which add multiple elements to a set or a hierarchy of sets simultaneously by passing an array of set element numbers created through the function `AimmsSetElementNumber`.

The functions

Set element representations

- `AimmsSetElementToOrdinal`,
- `AimmsSetElementToName`,
- `AimmsSetOrdinalToElement`,
- `AimmsSetOrdinalToName`,
- `AimmsSetNameToElement`, and
- `AimmsSetNameToOrdinal`

allow you to convert AIMMS' element numbers into ordinal numbers within a particular subset, and element names and vice versa. The functions will fail when the input representation does not correspond to an existing element.

In working with ordinal numbers, you should be aware that ordinal numbers are not invariant under changes to a set. When an element is added to or deleted from a set, or when the ordering of the set has changed, the ordinal numbers of some or all of its elements may have changed. In contrast, the element numbers and names of elements remain constant as long as the case used by the AIMMS model has not changed, or when the CLEANDEPENDENTS operator has not been applied to one or more root sets. You can verify the latter condition with a call to the function `AimmsIdentifierDataVersion` (see also Section 34.3).

Ordinal numbers may change

AIMMS represents the elements of a compound set by element numbers in a compound *root* set. With the functions

Compound elements

- `AimmsSetCompoundToTuple`, and
- `AimmsSetTupleToCompound`

you can obtain the tuple of element numbers in the corresponding simple root sets for a compound element number, and vice versa.

If a tuple does not correspond to an existing compound element, the function `AimmsSetTupleToCompound` will fail. In such a case, you can create a new compound element by calling the function `AimmsSetAddTupleToCompound` or the function `AimmsSetAddTupleToCompoundRecursive` instead, depending on whether you want to add the tuple to a compound root or subset, respectively. You can delete compound elements from a compound set using the function `AimmsSetDeleteElement`.

Adding and deleting compound elements

34.6 Executing AIMMS procedures

The AIMMS API allows you to execute procedures contained in the AIMMS model from within an external application. Both procedures with and without arguments can be executed, and scalar output results can be directly passed back to the external application. Table 34.5 lists the AIMMS API functions offered to obtain procedure handles, to execute AIMMS procedures or to schedule AIMMS procedures for later execution.

Running AIMMS procedures

With the function `AimmsProcedureHandleCreate` you can obtain a handle to a procedure with the given name within the model. In addition, AIMMS will return the number of arguments of the procedure, as well as the type of each argument. The possible argument types are:

Obtaining procedure handles

- one of the storage types *double*, *integer*, *binary* or *string* (discussed in Section 34.2) for scalar formal arguments, or
- a *handle* for non-scalar formal arguments.

<code>int AimmsProcedureHandleCreate(char *procedure, int *handle, int *nargs, int *argtype)</code>
<code>int AimmsProcedureHandleDelete(int handle)</code>
<code>int AimmsProcedureRun(int handle, int *argtype, AimmsValue *arglist, int *result)</code>
<code>int AimmsProcedureArgumentHandleCreate(int prochandle, int argnumber, int *arghandle)</code>
<code>int AimmsProcedureAsyncRunCreate(int handle, int *argtype, AimmsValue *arglist, int *request)</code>
<code>int AimmsProcedureAsyncRunDelete(int request)</code>
<code>int AimmsProcedureAsyncRunStatus(int request, int *status, int *result)</code>
<code>int AimmsExecutionInterrupt(void)</code>

Table 34.5: AIMMS API functions for execution requests

In addition to indicating the storage type of each argument, the `argtype` argument will also indicate whether an argument is input, output, or input-output. Through the function `AimmsProcedureHandleDelete` you can delete procedure handles created with `AimmsProcedureHandleCreate`.

You can use the function `AimmsProcedureRun` to run the AIMMS procedure associated with a given handle. If the AIMMS procedure has arguments, then you have to provide these, together with their types, through the `arglist` and `argtype` arguments. The (integer) return value of the procedure (see also pages 143 and 151) is returned through the `result` argument. If AIMMS is already executing another procedure (started by another thread), the call to `AimmsProcedureRun` blocks until the other execution request has finished. Section 34.10 explains how to prevent this blocking behavior by obtaining exclusive control over AIMMS.

Calling procedures

For each argument of the AIMMS procedure you have to provide both the type and value through the `argtype` and `arglist` arguments in the call to `AimmsProcedureRun`. You have the following possibilities.

Passing arguments

- If the argument is scalar, the argument type can be
 - the storage type returned by the function `AimmsProcedureHandleCreate`, in which case the argument value must be a pointer to a buffer of the indicated type containing the argument, or
 - a handle, in which case the argument value must be a handle associated with a scalar AIMMS identifier (slice) that you want to pass.
- If the argument is non-scalar, the argument type can only be a handle, and the argument value must be a handle corresponding to the identifier (slice) that you want to pass.

If you pass an argument as an identifier handle, this can either be a handle to a global identifier defined within the model, or a local argument handle obtained through a call to the function `AimmsProcedureArgumentHandleCreate` (see below).

When the input-output type of one or more of the arguments is `inout` or `output`, AIMMS will update the values associated with any handle argument, or, if a buffer containing a scalar value was passed, fill the buffer with the new value of the argument.

Output values

Through the function `AimmsProcedureArgumentHandleCreate` you can obtain a handle to the local arguments of procedures within your model. After creating these handles you can pass them as arguments to the function `AimmsProcedureRun`. The following rules apply.

Obtaining argument handles

- After creation, handles created by `AimmsProcedureArgumentHandleCreate` have no associated data.
- If the handle corresponds to an `Input` argument of the procedure, you can supply data prior to calling the procedure, and AIMMS will empty the handle after the execution of the procedure has completed.
- If the handle corresponds to an `InOut` or `Output` argument of the procedure, AIMMS will not empty the handle after completion of the procedure. If you want to supply data to a handle corresponding to an `InOut` argument in subsequent calls, you have to make sure to empty the handle (through the function `AimmsIdentifierEmpty`) prior to supplying the input data.

With the function `AimmsProcedureAsyncRunCreate` you can request asynchronous execution of a particular AIMMS procedure. The function returns an integer request handle for further reference. AIMMS will execute a requested procedure as soon as there are no other execution requests currently being executed or waiting to be executed. *Note that you should make sure that the `AimmsValue` array passed to AIMMS stays alive during the asynchronous execution of the procedure.* Failure to do so, may result in illegal memory references during the actual execution of the AIMMS procedure. This is especially true when the array contains references to scalar integer, double or string `InOut` or `Output` buffers within your application to be filled by the AIMMS procedure.

Requesting asynchronous execution

Through the function `AimmsProcedureAsyncRunStatus` you can obtain the status of an outstanding asynchronous execution request. The status of such a request can be

Obtaining the status

- `pending`,
- `running`,
- `finished`,
- `deleted`, or
- `unknown` (for an invalid request handle).

When the request is in the `finished` state, the return value of the AIMMS procedure will be returned via the `result` argument.

You should make sure to delete all asynchronous execution handles requested during a session using the function `AimmsProcedureAsyncRunDelete`. *Failure to delete all finished requests may result in a serious memory leak if your external DLL generates many small asynchronous execution requests.* If you delete a pending request, AIMMS will remove the request from the current execution queue. The function will fail if you try to delete a request that is currently being executed.

Deleting a request

When an AIMMS procedure has been started by a separate thread in your program you can interrupt it using the function `AimmsExecutionInterrupt`. This function returns `AIMMSAPI_SUCCESS` when AIMMS was idle and `AIMMSAPI_FAILURE` was executing a procedure.

Interrupting an existing run

34.7 Passing errors and messages

The AIMMS API functions in Table 34.6 let you send error and warning messages to AIMMS and get the current AIMMS status. In addition, you can obtain the error number and description of the last AIMMS API error.

Passing errors and messages

<code>int AimmsAPIPassMessage(int severity, char *message)</code>
<code>int AimmsAPIStatus(int *status)</code>
<code>int AimmsAPILastError(int *code, char *message)</code>

Table 34.6: AIMMS API functions for error messages

With the function `AimmsAPIPassMessage` you can send error and warning messages to the end-user of your DLL in AIMMS. Such errors and warnings are displayed to the end-user in the AIMMS message window. For every message you must indicate a severity code, the complete list of which is included in the `aimmsapi.h` header file. When AIMMS receives a message with error severity, a run-time error is generated. The end-user of an application can set execution options to filter out those warning messages which are below a certain severity threshold.

Passing errors and messages

If a function in your DLL is called from within an AIMMS project, and you want to pass back an error message to the model without automatically opening the AIMMS message window, you should not use the function `AimmsAPIPassMessage`, but instead assign the message to the predefined AIMMS string parameter `CurrentErrorMessage`. To assign a value to it, you should create a handle to it via the function `AimmsIdentifierHandleCreate` and assign the message using the function `AimmsValueAssign`. It is then up to the model developer calling your function, whether the message stored in `CurrentErrorMessage` should be displayed (e.g. in the AIMMS message window).

Setting CurrentErrorMessage

Through the function `AimmsAPIStatus` you can obtain the current status of the AIMMS execution engine, such as executing, solving, ready, etc. The complete list of possible status codes and their meaning is included in the `aimmsapi.h` header file.

Obtaining the execution status

Whenever a call to an AIMMS API function fails, the function returns `AIMMS-API_FAILURE` as its return value. In such a case, you can obtain the precise error code and a message describing the error through the function `AimmsAPILastError`. The complete list of error codes is contained in the `aimmsapi.h` header file. By modifying the API-related execution options, you can also enforce that every API error is listed in the AIMMS message window.

Obtaining API errors

34.8 Raising and handling errors

The error passing described in the previous section is retained in AIMMS 3 in order not to break existing applications. The use of the error handling described in this section, however, is encouraged as it is more in line with the error handling framework described in Section 8.4. In addition, all errors, including all their parts, can be retrieved. The AIMMS API functions in Table 34.7 enable the raising and handling of errors and retrieving the current AIMMS status.

Raising and handling errors

<code>int AimmsErrorStatus(void)</code>
<code>int AimmsErrorCount(void)</code>
<code>char *AimmsErrorMessage(int errNo)</code>
<code>int AimmsErrorSeverity(int errNo)</code>
<code>char *AimmsErrorCode(int errNo)</code>
<code>char *AimmsErrorCategory(int errNo)</code>
<code>int AimmsErrorNumberOfLocations(int errNo)</code>
<code>char *AimmsErrorFilename(int errNo)</code>
<code>char *AimmsErrorNode(int errNo, int pos)</code>
<code>char *AimmsErrorAttributeName(int errNo, int pos)</code>
<code>int AimmsErrorLine(int errNo, int pos)</code>
<code>int AimmsErrorColumn(int errNo)</code>
<code>time_t AimmsErrorCreationTime(int errNo)</code>
<code>int AimmsErrorDelete(int errNo)</code>
<code>int AimmsErrorClear(void)</code>
<code>int AimmsErrorRaise(int severity, char *message, char *code)</code>

Table 34.7: AIMMS Raising and handling errors in the AIMMS API

The functions `AimmsErrorStatus`, `AimmsErrorCount`, `AimmsErrorGet`, `AimmsErrorDelete` and `AimmsErrorClear` all manipulate the global error collector. The global error collector is described in Section 8.4. The function `AimmsErrorStatus` scans the contents of the global error collector and returns

Global error collector manipulation

- `AIMMSAPI_SEVERITY_CODE_NEVER`: if the global error collector is empty,

- **AIMMSAPI_SEVERITY_CODE_WARNING**: if it contains only warnings, or
- **AIMMSAPI_SEVERITY_CODE_ERROR**: if it contains at least one error.

The function `AimmsErrorCount` does not return a status code, instead it directly returns the number of errors and warnings in the global error collector. With the functions `AimmsErrorMessage`, `AimmsErrorSeverity`, `AimmsErrorCategory`, `AimmsErrorCode`, `AimmsErrorNumberOfLocations`, `AimmsErrorLine`, `AimmsErrorNode`, `AimmsErrorAttributeName`, `AimmsErrorFilename`, `AimmsErrorColumn`, and `AimmsErrorCreationTime` actual error information is obtained. In these functions the `errNo` argument should be in the range `{1..AimmsErrorCount()}` and the `pos` argument should be in the range `{1..AimmsErrorNumberOfLocations(errNo)}`.

None of the AIMMS API functions throws an exception, nor do any of them let an exception pass through. The example below serves as a simple template to call `AimmsProcedureRun` and handle all errors occurring during that execution run.

Example for API calls

```
int ErrCount, errNo, apr_stat ;

apr_stat = AimmsProcedureRun( procHandle, ... );
ErrCount = AimmsErrorCount();
if ( ErrCount ) {
    for ( errNo = 1 ; errNo <= ErrCount ; errNo ++ ) {

        // Handle the error; replace the next line as
        // appropriate for the application at hand.
        printf( "Error %d: %s\n", errNo, AimmsErrorMessage(errNo) );

    }
    AimmsErrorClear();
} else if ( apr_stat == AIMMSAPI_FAILURE ) {
    printf("Aimms failed for an unknown reason.\n");
}
```

The function `AimmsErrorRaise(severity, message, code)` can raise errors and warnings. These errors will be handled by the currently active error handler as described in Section 8.4. If there is no currently active error handler, the error is directly placed in the global error collector. The call to this function is similar to the RAISE statement, see Section 8.4.2. The code argument is optional. The severity argument should be either

Raising an error

- **AIMMSAPI_SEVERITY_CODE_WARNING**: indicating a warning or
- **AIMMSAPI_SEVERITY_CODE_ERROR**: indicating an error.

The category of an error raised by `AimmsErrorRaise` is fixed to 'User'.

34.9 Opening and closing a project

The AIMMS API functions in Table 34.8 allow you to open and close an AIMMS project from within your own application.

Opening and closing a project

<code>int AimmsProjectOpen(char *commandline, int *handle)</code>
<code>int AimmsServerProjectOpen(char *commandline, int *handle)</code>
<code>int AimmsProjectClose(int handle, int interactive)</code>
<code>int AimmsProjectWindow(HWND *window)</code>

Table 34.8: AIMMS API functions for opening and closing projects

If you want to use AIMMS as an optimization engine from within an external program, you can use the function `AimmsProjectOpen` to open the AIMMS project which contains the model that you want to connect to. To open an AIMMS project you must specify the command line containing the project file name as well as any other command line options with which you want to run AIMMS, *but without the name of the AIMMS executable*. If the project is not in the current working directory, the directory in which the project is contained must be appended to the project file name. On success, you obtain a project handle which must be used to close the project. Because a single AIMMS instance can only run a single project, the function fails if a project is already running in this AIMMS instance.

Opening a project

When you are running an AIMMS project as a server application, you do not want windows or message boxes to appear on the server desktop under any circumstances. Although you can open an AIMMS project in a minimized or hidden fashion through commandline arguments passed to the `AimmsProjectOpen` function, AIMMS can still present some message boxes, e.g. to report licensing problems during startup. With the function `AimmsServerProjectOpen` you can open an AIMMS project absolutely without any windowing support. If AIMMS encounters any problems during startup, the function will fail and you can retrieve the error message through the function `AimmsAPILastError`.

Opening a server project

When you open an AIMMS project from within your own application through the AIMMS API, the normal AIMMS licensing arrangements apply. When no valid AIMMS license is available on the host computer, a call to either `AimmsProjectOpen` or `AimmsServerProjectOpen` will fail.

License required

With the function `AimmsProjectClose` you can request AIMMS to close the current project, and, subsequently, to terminate itself. With the interactive argument you can indicate whether the project must be closed in an interactive manner (i.e. whether the user must be able to answer any additional dialog box that may appear), or that the default response is assumed. The request will fail if the project handle is not equal to the project handle returned by the function `AimmsProjectOpen`, thus disallowing you to close a project that was not opened by yourself.

Closing a project

Through the function `AimmsProjectWindow` you can obtain the WIN32 window handle associated with the current AIMMS project. You can use the window handle in any WIN32 function call inside your DLL that requires the AIMMS window handle to function properly.

Obtain the AIMMS window handle

34.10 Thread synchronization

The AIMMS API allows multiple DLLs to be active within the context of a single project. While some of these DLLs may only be useful when called from within your AIMMS project itself, you may want other DLLs to run independently in a separate thread of execution. Such behavior may be necessary, for instance, when

Multiple threads

- you want to link AIMMS to an online data source, where an independent DLL collects the online data and passes it on to AIMMS whenever appropriate, or
- you want to call AIMMS as an independent optimization engine from within your own program and need to pass data to AIMMS whenever necessary.

When you open an AIMMS project by calling the function `AimmsProjectOpen` from within your own application, AIMMS will create a new thread. This AIMMS thread will deal with

AIMMS thread

- all end-user interaction initiated from within the AIMMS end-user interface (which is created as part of opening the project), and
- all asynchronous execution requests that are initiated either from within your application, another external DLL linked to your AIMMS project, or from within the model itself.

Whenever you want to call AIMMS API functions from within a thread started by yourself, you must make sure that the thread is well-equipped to do so by calling the AIMMS thread (un)initialization functions listed in Table 34.9.

Thread initialization

```
int AimmsThreadAttach(void)
int AimmsThreadDetach(void)
```

Table 34.9: AIMMS API functions for thread initialization

Prior to calling any other AIMMS API function from within a newly created thread, you should call the function `AimmsThreadAttach`. It will make sure that any thread-specific initialization required for calling the AIMMS execution engine is performed properly. Similarly, you should call the function `AimmsThreadDetach` just prior to exiting the thread.

Initializing threads...

Among others, a call to `AimmsThreadAttach` will initialize the Microsoft COM library in a manner compatible with the COM apartment model employed by AIMMS. Therefore, if you are using COM interfaces within your thread, you should not call the COM SDK functions `CoInitialize` (or `CoInitializeEx`) and `CoUninitialize` directly to initialize the COM library, but rather call `AimmsThreadAttach` and `AimmsThreadDetach`.

... includes COM initialization

Whenever an AIMMS project runs in a multi-threaded environment, synchronization of the execution and data retrieval requests becomes of the utmost importance. By default, AIMMS will make sure that no two execution or data retrieval requests initiated from different threads are dealt with simultaneously. However, this default synchronization scheme does not preclude that the execution of two subsequent requests from one thread is interrupted by a request from another thread.

Thread synchronization

When the proper functioning of your application requires that your execution and data retrieval requests to AIMMS are not interrupted by requests from competing threads, you can use the functions listed in Table 34.10 to obtain exclusive control over the AIMMS execution engine.

Obtaining exclusive control

```
int AimmsControlGet(int timeout)
int AimmsControlRelease(void)
```

Table 34.10: AIMMS API functions for obtaining exclusive control

With the function `AimmsControlGet` you can restrict control over the current AIMMS session exclusively to the thread calling `AimmsControlGet`. Execution and data retrieval requests from any thread other than this controlling thread (including the AIMMS thread itself) will block until the controlling thread has released the control. The function `AimmsControlRelease` releases the exclusive control over the AIMMS session. *Note that every successful call to `AimmsControlGet` must be followed by a corresponding call to `AimmsControlRelease`, or AIMMS*

Obtaining and releasing control

will be inaccessible to all other threads for the remainder of the session. `AimmsControlRelease` fails when the calling thread does not have exclusive control.

When another thread has exclusive control over AIMMS, either obtained explicitly through a call to `AimmsControlGet` or implicitly through an execution or data retrieval request, the function `AimmsControlGet` will block *timeout* milliseconds before returning with a failure. By choosing a *timeout* of `WAIT_INFINITE`, the function `AimmsControlGet` will block until it gets exclusive control.

Waiting for the control

If you want to make sure that a subsequent execution request will never block, you can

Nonblocking execution

- call `AimmsControlGet` with a timeout of 0 milliseconds,
- perform the execution request when successful, and
- subsequently release the control.

The call to `AimmsControlGet` has the effect of verifying that no other thread is using AIMMS at the moment. If you cannot get exclusive control, you must store the request for later execution.

34.11 Interrupts

During the execution of a procedure in your model, the AIMMS thread will block. This effectively prevents you from interrupting the AIMMS execution, for instance, to update data in the model, or just to abort the current procedure execution. Likewise, while a function in your DLL that was executed from within an AIMMS procedure is still running, AIMMS cannot service any end-user requests. The functions listed in this section allow your DLL and AIMMS to work together in a cooperative manner in such situations.

Interrupts

You can use the functions listed in Table 34.11 to handle two-way interrupts.

Handling interrupts

<pre>int AimmsInterruptCallbackInstall(AimmsInterruptCallback cb) int AimmsInterruptPending(void)</pre>

Table 34.11: AIMMS API functions for handling interrupts

With the function `AimmsInterruptCallbackInstall` you can pass a function pointer with a prescribed prototype to AIMMS, which AIMMS will call on a regular basis during subsequent execution of an AIMMS procedure. Note that the installed callback is *thread-local*, i.e., AIMMS will only call the callback procedure from within an AIMMS procedure that is executing in the same thread in which you called `AimmsInterruptCallbackInstall` to install the callback.

Installing a callback

Within a callback function AIMMS allows you to request or modify model data, or to run model procedures, which would normally be prohibited because calls to the AIMMS API block when AIMMS is executing (see also Section 34.10). Through the argument of the callback function AIMMS passes its current state (just executing, or within a solve), while you can indicate, through the return value of the callback function, whether you

*Within a
callback*

- want to interrupt the current solve but continue the remainder of the current execution,
- want to interrupt the current execution all together, or
- do not want to interrupt the current execution at all.

Because AIMMS will call a callback procedure quite regularly, it is advisory to keep the actions executed within it to a minimum, or AIMMS could be slowed down unacceptably.

You can uninstall a previously installed callback function by simply calling the function `AimmsInterruptCallbackInstall` with a null pointer as the callback function argument. Note that it is even possible to uninstall a callback function—or modify a callback function—during a call (by AIMMS) to the currently installed callback function.

*Uninstalling the
callback*

When AIMMS calls a function within an external DLL, this would normally prevent AIMMS from servicing end-user requests to update end-user pages, modify model data, or even to interrupt the execution of the current AIMMS execution, i.e. the execution of your function. This is not a problem when a call to your function only takes a small amount of time to execute, but might be unacceptable when your function takes a long time to complete. In such situations, you might consider to insert calls to the function `AimmsInterruptPending` at strategic places in your source code. With it, you allow AIMMS to service such requests, and to call any callback functions installed by other DLLs. On return, `AimmsInterruptPending` returns

*Pending
interrupts*

- `AIMMSAPI_TRUE` when AIMMS received a request to interrupt the current execution, or
- `AIMMSAPI_FALSE` when there was no interrupt request.

When an interrupt was requested you should abort the execution of your external function as soon as possible.

34.12 Model Edit Functions

The AIMMS API supports Model Edit Functions allowing external applications to inspect, modify, or even construct AIMMS models. In this section, the model edit functions are introduced using a small example. Subsequently, after briefly describing the relation to runtime libraries plus the conventions

*AIMMS Model
Edit Functions*

used, several tables containing model edit functions, are presented and described. Finally, the limitations of using model edit functions through the AIMMS API are described briefly.

In the following example, an element parameter `nextCity` is created with a simple definition. Model editing is done using **model editor** handles. These handles provide access to the identifiers in the model, and should not be confused with the data handles and procedure handles described elsewhere in this chapter.

Small example

The model editor handle `int dsMEH` refers to a declaration section, whereas the model editor handle `int ncMEH` refers to the parameter `nextCity`.

```
1 AimmsMeCreateNode("nextCity", AIMMSAPI_ME_IDTYPE_ELEMENT_PARAMETER, dsMEH, 0, &ncMEH);
2 AimmsMeSetAttribute(ncMEH, AIMMSAPI_ME_ATTR_INDEX_DOMAIN, "i");
3 AimmsMeSetAttribute(ncMEH, AIMMSAPI_ME_ATTR_RANGE, "Cities");
4 AimmsMeSetAttribute(ncMEH, AIMMSAPI_ME_ATTR_DEFINITION,
   "if i == last(Cities) then first(Cities) "
   "else Element(Cities,ord(i)+1) endif");
5 AimmsMeCompile(ncMEH);
6 AimmsMeCloseNode(ncMEH);
```

A line by line explanation of this example follows below. For the sake of brevity, error handling, such as suggested in Section 34.8, is omitted here.

- **Line 1:** Creates an element parameter named `nextCity`. The fourth argument of `AimmsMeCreateNode` is the position within the section. A 0 indicates that it should be placed at the end of the section.
- **Lines 2-4:** Set the attributes `IndexDomain`, `Range`, and `Definition` of this parameter. Note that only the text is passed, these calls do not use the AIMMS compiler to compile them.
- **Line 5:** Compiles the element parameter `nextCity`. Only now is the text of the attributes actually checked and compiled.
- **Line 6:** The function `AimmsMeCloseHandle` de-allocates the handle `ncMEH` but the created identifier `nextCity` remains in the model.

Section 35.6 describes the model editing facility available in the AIMMS language using runtime libraries. The advantage of using the AIMMS API, instead of runtime libraries, for model editing is that the entire model can be edited, including the main model, provided that there is no AIMMS procedure active while executing a model edit function from within the AIMMS API. The cost is that multiple languages have to be used.

Relation to runtime libraries

The model edit functions follow the following conventions:

- Each function starts with `AimmsMe`.
- These functions return either `AIMMSAPI_SUCCESS` or `AIMMSAPI_FAILURE`.
- A model editor handle `MEH` has to be closed by either `AimmsMeCloseNode` or `AimmsMeDestroyNode`.

Conventions for model edit functions

- No distinction is made between identifiers and nodes in the model editor tree, they are both called "nodes".
- String output arguments use the type `AimmsString` as is explained in Section 34.2.

Table 34.12 lists the functions available for manipulating model editor roots. The number of roots available for model editing is stored by the function `AimmsMeRootCount(count)` in its output argument `count`. Note that `count` is always at least 1, since there is always the main model. The root `Predeclared identifiers` is not included in this count. As the root `Predeclared identifiers` and its sub-nodes cannot be changed, it is not included in this count. To obtain a handle for an existing root, the function `int AimmsMeOpenRoot(pos, MEH)` can be used. A model editor handle is then created and stored in `MEH`. If `pos` is 0, the main model root is opened. If `pos` is in the range `{ 1 .. count-1 }` then a library is opened. If `pos` equals `count` then the `predeclared root Predeclared identifiers` is opened. The root `Predeclared identifiers` and its sub-nodes are read only. In order to create a new runtime library the function `AimmsMeCreateRuntimeLibrary(name, prefix, MEH)` can be used. The position of the new library is at the end of the existing libraries.

Model roots

<code>int AimmsMeRootCount(int *count)</code>
<code>int AimmsMeOpenRoot(int pos, int *MEH)</code>
<code>int AimmsMeCreateRuntimeLibrary(char *name, char *prefix, int *MEH)</code>
<code>int AimmsMeNodeExists(char*name, int nMEH, int *exists)</code>
<code>int AimmsMeOpenNode(char*name, int nMEH, int *MEH)</code>
<code>int AimmsMeCreateNode(char *name, int idtype, int pMEH, int pos, int *MEH)</code>
<code>int AimmsMeCloseNode(int MEH)</code>
<code>int AimmsMeDestroyNode(int MEH)</code>

Table 34.12: Model edit functions for roots and nodes

The function `AimmsMeNodeExists(name, nMEH, exists)` can be used to test if an identifier exists. This function returns `AIMMSAPI_FAILURE` when `nMEH` does not indicate a valid namespace, or when `name` is not a valid identifier name. If the name is a declared identifier in namespace `nMEH`, then `exists` is set to 1, and if not to 0. The function `AimmsMeOpenNode(name, nMEH, MEH)` creates a handle to the node with name `name` in the namespace determined by the model editor handle `nMEH`. If successful, a model editor handle is created and stored in the output argument `MEH`. If `nMEH` equals `AIMMSAPI_NULL_HANDLE_NUMBER`, then the namespace of the main model is used. A new node with name `name` and type `idtype` can be created using the function `AimmsMeCreateNode(name, idtype, pMEH, pos, MEH)`. The value of `idtype` must be one of the constants defined in `aimmsapi.h` starting with `AIMMSAPI_ME_IDTYPE_`. The parent node of the new node is determined by the model editor handle `pMEH`. The value `pos` determines the new position of the node within the parent node. If `pos` is outside the range of existing children `{1..n}`, the new identifier is placed at the end, otherwise the

Opening or creating a node

existing children at positions `pos .. n` are shifted to positions `pos+1 .. n+1` where `n` was the old number of children of `pMEH`.

Table 34.12 not only lists the functions to open or create nodes, but also shows the complementary functions to close or destroy nodes. The function `AimmsMeCloseNode(MEH)` de-allocates the handle `MEH` but leaves the corresponding node in the model intact. The function `AimmsMeDestroyNode(MEH)` destroys the node corresponding to `MEH` and all nodes below that node in the model, and subsequently deallocates the handle `MEH`.

Closing or destroying a node

Table 34.13 lists the functions that return the name of a node. The function `AimmsMeName(MEH, name)` stores the name of the node to which `MEH` refers without any prefixes in the output argument `name`. The function `AimmsMeRelativeName(MEH, rMEH, rName)` stores the name of `MEH` such as it should be used from within the node `rMEH` in the output argument `rName`. A fully qualified name is stored in `rName` when `MEH` is the `AIMMSAPI_ME_NULL_HANDLE_NUMBER` handle.

The name of a node

<code>int AimmsMeName(int MEH, AimmsString *name)</code>
<code>int AimmsMeRelativeName(int MEH, int rMEH, AimmsString *rName)</code>
<code>int AimmsMeType(int MEH, int *meType)</code>
<code>int AimmsMeTypeName(int typeNo, AimmsString *tName)</code>
<code>int AimmsMeAllowedChildTypes(int MEH, int *typeBuf, int typeBufsize, int *maxTypes)</code>

Table 34.13: Model edit functions for name and type

In addition, Table 34.13 lists the functions for the type of a node. The function `AimmsMeType(MEH, meType)` stores the type of the node `MEH` in the output argument `meType`. The value of `meType` refers to one of the constants in `aimmsapi.h` starting with `AIMMSAPI_ME_IDTYPE_`. The function `AimmsMeAllowedChildTypes(MEH, typeBuf, typeBufsize, maxTypes)` stores the types of children allowed below the node `MEH` in the buffer `typeBuf` while respecting its size `typeBufsize`. The maximum number of child types below `MEH` is stored in the output argument `maxTypes`. The utility function `AimmsMeTypeName(typeNo, tName)` stores the name of the type `typeNo` in the output argument `tName`.

The type of a node

<code>int AimmsMeGetAttribute(int MEH, int attr, AimmsString *text)</code>
<code>int AimmsMeSetAttribute(int MEH, int attr, const char *txt)</code>
<code>int AimmsMeAttributes(int MEH, int attrsBuf[], int attrBufSize, int *maxNoAttrs)</code>
<code>int AimmsMeAttributeName(int attr, AimmsString *name)</code>

Table 34.14: Model edit functions for attributes

Table 34.14 lists the functions available for handling the attributes of a node. All attributes correspond to constants in the `aimmsapi.h` file. These constants start with `AIMMSAPI_ME_ATTR_`. The function `AimmsMeGetAttribute(MEH, attr, text)` stores the contents of attribute `attr` of node `MEH` in the output argument `text`. The function `AimmsMeSetAttribute(MEH, attr, txt)` sets the contents of attribute `attr` of node `MEH` to `txt`. This function will fail if attribute `attr` is not applicable to identifier `MEH`, but the text itself is not checked for errors. The function `AimmsMeAttributes(MEH, attrsBuf, attrBufSize, maxNoAttrs)` provides the applicable attributes for these two functions. It will store the constants corresponding to the attributes available to node `MEH` in `attrsBuf` while respecting the size of that buffer `attrBufSize`. The maximum number of attributes available to node `MEH` is stored in `maxNoAttrs`. The function `AimmsMeAttributeName(attr, name)` stores the name of `attr` in `name`.

The attributes of a node

<pre>int AimmsMeNodeRename(int MEH, char *newName) int AimmsMeNodeMove(int MEH, int pMEH, int pos) int AimmsMeNodeChangeType(int MEH, int newType) int AimmsMeNodeAllowedTypes(int MEH, int* typeBuf, int typeBufsize, int *maxNoTypes)</pre>

Table 34.15: Model edit functions for node manipulations

The functions that support changing the aspects of a node such as name, location, and type of a node are also shown in Table 34.15. The function `AimmsMeNodeRename(MEH, newName)` changes the name of a node, and the name-change is applied to the attribute texts that reference this node. An entry is appended to the name change file if the node is not a runtime node. The function `AimmsMeNodeMove(MEH, pMEH, pos)` moves the node `MEH` to child position `pos` of node `pMEH`. If this results in a change of namespace, the corresponding namechange is applied to the attributes that reference this node. In addition, an entry is appended to the corresponding name change file if this node is not a runtime node. Moves from one library to another are not supported, nor is a move in or out of the main model. The function `AimmsMeNodeChangeType(MEH, newType)` changes the type of a node. It will retain available attributes whenever possible. The function `AimmsMeNodeAllowedTypes` can be used to query which types, if any, a particular node can be changed to. The function `AimmsMeNodeAllowedTypes(MEH, typeBuf, typeBufsize, maxNoTypes)` will store all the types into which node `MEH` can be changed in a buffer `typeBuf` that respects the size `typeBufsize`. The maximum number of types into which `MEH` can be changed is stored in `maxNoTypes`.

Basic node manipulations

Table 34.16 lists the functions that permit walking all nodes in the model editor tree. The function `AimmsMeParent(MEH, pMEH)` creates a model editor handle to the parent of `MEH`, and stores this handle in the output argument `pMEH`. The function `AimmsMeFirst(MEH, fMEH)` creates a model editor handle to the first child of `MEH`, and stores this handle in the output argument `fMEH`. The function

Tree walk of the model

`AimmsMeNext(MEH, nMEH)` creates a model editor handle to the node next to MEH, and stores this handle in the output argument nMEH. If such a parent, first child, or next node does not exist the `AIMMSAPI_ME_NULL_HANDLE_NUMBER` handle is stored in the output argument although the corresponding function does not fail.

<pre>int AimmsMeParent(int MEH, int *pMEH) int AimmsMeFirst(int MEH, int *fMEH) int AimmsMeNext(int MEH, int *nMEH) int AimmsMeImportNode(int MEH, char *fn, const char *pwd) int AimmsMeExportNode(int MEH, char *fn, const char *pwd)</pre>

Table 34.16: Reading, writing and tree walking a model editor tree

The functions that allow the reading of an AIMMS section from a file, or writing a section to a file are also listed in Table 34.16. They use the Text .ams file format. The function `AimmsMeImportNode(MEH, fn, pwd)` reads a file fn and stores the resulting model structure at node MEH. The function `AimmsMeExportNode(MEH, fn, pwd)` writes the model structure at node MEH to file fn. If MEH does not refer to an AIMMS section, module, library, or model, the functions `AimmsMeImportNode` and `AimmsMeExportNode` will fail.

Reading and writing (portions of) a model

The model edit functions available for compilation and model status queries are listed in Table 34.17. The central function `AimmsMeCompile (MEH)` compiles the node MEH and all its sub-nodes. The entire application (main model and libraries) is compiled if the argument MEH equals `AIMMSAPI_ME_NULL_HANDLE_NUMBER`. If this compilation step is successful then the procedures are runnable. The function `AimmsMeIsRunnable(MEH, r)` stores 1 in the output argument r if the procedure referenced by MEH is runnable. The function `AimmsMeIsReadOnly(MEH, r)` stores 1 in the output argument r if the node resides in a read-only library, such as the predeclared identifiers, or a library that was read from a read only file.

Compilation

<pre>int AimmsMeCompile(int MEH) int AimmsMeIsRunnable(int MEH, int *r) int AimmsMeIsReadOnly(int MEH, int *r)</pre>
--

Table 34.17: Model edit functions for compilation and status queries

The following limitations apply to model edit functions from within the AIMMS API:

Limitations

1. Compound sets are not supported.
2. The `SourceFile` attribute is not supported.
3. The current maximum number of identifiers is thirty thousand.

Further, when an AIMMS procedure is running, the identifiers in the main application can not be modified as explained in [Section 35.6](#).