ECL^{i}PS^e

Embedding and Interfacing Manual

Release 7.0

Stefano Novello (IC-Parc)
Joachim Schimpf (IC-Parc)
Kish Shen (IC-Parc)
Josh Singer (Parc Technologies Ltd.)

February 26, 2020

Trademarks

Contents

Chapter 1

Introduction

This manual contains the information needed to interface ECL^{i}PS^e code to C, C++ or Java environments, or to use it from within scripting languages such as Tcl. ECL^{i}PS^e is available in the form of a linkable library, and a number of facilities are available to pass data between the different environments, to make the integration as close as possible.
It also contains information on interfacing to external software systems, such as database management systems.
Example sources can be found in the ECL^{i}PS^e installation directory under **doc/examples**.

Chapter 2

Calling ECL^{i}PS^e from C++

This chapter describes how ECL^{i}PS^e can be included in an external program as a library, how to start it, and how to communicate with it. Code examples are given in C++. For the equivalent C functions, please refer to chapter C.

2.1 To get started

This section is about the prerequisites for working with ECL^{i}PS^e in your development environment. The directory structure, the libraries and the include files are described.

2.1.1 Directories

The libraries and include files needed to use ECL^{i}PS^e as an embedded component are available under the ECL^{i}PS^e directory which was set-up during installation. If you have access to a stand-alone ECL^{i}PS^e it can be found using the following query at the ECL^{i}PS^e prompt:

```
[eclipse 1]: get_flag(installation_directory,Dir).

Dir = "/usr/local/eclipse"
yes.
[eclipse 2]
```

We will assume from here that ECL^{i}PS^e was installed in "/usr/local/eclipse".
You would find the include files in "/usr/local/eclipse/include/$ARCH" and the the libraries in "/usr/local/eclipse/lib/$ARCH" where "$ARCH" is a string naming the architecture of your machine. This can be found using the following ECL^{i}PS^e query:

```
[eclipse 2]: get_flag(hostarch,Arch).

Arch = "sun4"
yes.
[eclipse 3]:
```

You will need to inform your C or C++ compiler and linker about these directories so that these tools can include and link the appropriate files. A make file "Makefile.external" can be found

together with the libraries. The definitions in that makefile may have to be updated according to your operating system environment.
A set of example C and C++ programs can be found in "/usr/local/eclipse/doc/examples".
When delivering an application you will have to include with it the contents of the directory "/usr/local/eclipse/lib" without which ECL^{i}PS^e cannot work. Normally this would be copied into the directory structure of the delivered application. The interface can set different values for this directory, enabling different applications to have different sets of libraries.

2.1.2 Definitions

To include the definitions needed for calling the ECL^{i}PS^e library in a C program use:

```
#include <eclipse.h>
```

For C++ a more convenient calling convention can be used based on some classes wrapped around these C definitions. To include these use:

```
#include <eclipseclass.h>
```

2.1.3 Compiling, linking and running on Unix/Linux

Assuming that the environment variable ECLIPSEDIR is set to the ECL^{i}PS^e installation directory and the environment variable ARCH is set to the architecture/operating system name, an application can be built as follows:

```
gcc -I$ECLIPSEDIR/include/$ARCH eg_c_basic.c -L$ECLIPSEDIR/lib/$ARCH -leclipse
```

This will link your application with the shared library `libeclipse.so`.
At runtime, your application must be able to locate `libeclipse.so`. This can be achieved by adding *ECLIPSEDIR*/*lib*/ARCH to your `LD_LIBRARY_PATH` environment variable.
The embedded ECL^{i}PS^e finds its own support files (e.g. ECL^{i}PS^e libraries) through the `ECLIPSEDIR` environment variable. This must be set to the location where ECL^{i}PS^e is installed, e.g. `/usr/local/eclipse`. Alternatively, the application can invoke `ec_set_option` to specify the `ECLIPSEDIR` location before initialising the embedded ECL^{i}PS^e with `ec_init`.

2.1.4 Compiling, linking and running on Windows

If you use GCC, you can link either directly against `eclipse.dll` or against `eclipse.dll.a`. The required command line is similar to the Unix case.
If you use a Microsoft compiler, make sure you have the following additional settings in your C/C++ compiler/development system:

- In the **C/C++ Preprocessor** settings, specify the ECL^{i}PS^e include directory as an additional include directory, e.g. `C:\Program Files\ECLiPSe 5.10\include\i386_nt`.
- In the **Link** settings, specify `eclipse.lib` as an additional object/library module, and the location of this library, e.g. `C:\Program Files\ECLiPSe 5.10\lib\i386_nt` as an additional library path.

Moreover, you need to create an `eclipse.lib` for the compiler to link against. This file can be created from `eclipse.def` and `eclipse.dll` (which are part of the ECL^{i}PS^e distribution), using the `lib.exe` or `link.exe` tool (which comes with the C/C++ development system).

```
cd C:\Program Files\ECLiPSe 5.10\lib\i386_nt
lib.exe /def:eclipse.def
```

Warnings about import directives can be ignored. If you do not have `lib.exe`, try instead

```
link.exe /lib /def:eclipse.def
```

At runtime, your application must be able to locate `eclipse.dll`, i.e. you should either

- copy `eclipse.dll` into the folder where your application is located, or
- copy `eclipse.dll` into one of Windows' standard library folders, or
- add the path to the folder where `eclipse.dll` can be found to your PATH environment variable.

The `eclipse.dll` finds its own support files (e.g. ECL^{i}PS^e libraries) through the `ECLIPSEDIR` registry entry under the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\IC-Parc\ECLiPSe\X.Y` (X.Y is the version number). This must be set to the location where ECL^{i}PS^e is installed, e.g. `C:/Eclipse`. Alternatively, the application can invoke `ec_set_option` to specify the `ECLIPSEDIR` location before initialising the embedded ECL^{i}PS^e with `ec_init`.

2.2 Creating an ECL^{i}PS^e context

ECL^{i}PS^e runs as a special thread (we will call it ECL^{i}PS^e engine) within your application, maintaining its own execution state. This section is about when and how to initialise it. There are parameters to be applied before initialisation, but these are usually acceptable. These parameters are described in Appendix A.
Although it is useful to think of ECL^{i}PS^e as a thread, it is not an operating system thread, but is rather implemented as a set of C functions that maintain some state. This state is the complete execution state of the ECL^{i}PS^e engine, its stack of goals, its success and failure continuations and its global stack of all constructed data.
At particular points during its execution ECL^{i}PS^e will yield control back to the C level. This is implemented by returning from a function. ECL^{i}PS^e can then be restarted from the exact point it left off. This is implemented by a function call.

2.2.1 Initialisation

You initialise ECL^{i}PS^e by calling the parameterless function

```
int ec_init();
```

A process should do this just once. `ec_init` returns 0 on success or -1 if an error occurred. It is possible to influence the initialisation of ECL^{i}PS^e by setting initialisation options as described in Appendix A.
None of the functions of the interface work before this initialisation. In particular in C++, if you use static variables which are constructed by calling ECL^{i}PS^e functions you must arrange for the initialisation to occur before the constructors are called.

2.3 Control flow

ECL^{i}PS^e and a C or C++ main program are like threads running in a single process. Each maintains its state and methods for exchanging data and yielding control to the other thread. The main method of sending data from C++ to ECL^{i}PS^e is by posting goals for it to solve. All posted goals are solved in conjunction with each other, and with any previously posted goals that have succeeded.
Data is passed back by binding logical variables within the goals.
Control is explicit in C++. After posting some goals, the C++ program calls the `EC_resume()` function and these goals are all solved. A return code says whether they were successfully solved or whether a failure occurred.
In ECL^{i}PS^e control is normally implicit. Control returns to C++ when all goals have been solved.

```
#include        "eclipseclass.h"

main()
{
    ec_init();

    /* writeln("hello world"), */
    post_goal(term(EC_functor("writeln",1),"hello world"));
    EC_resume();
    ec_cleanup(0);
}
```

The above is an example program that posts a goal and executes it.

2.3.1 Control flow and search

Using this model of communication it is possible to construct programs where execution of C++ code and search within the ECL^{i}PS^e are interleaved.
If you post a number of goals (of which some are non-deterministic) and resume the ECL^{i}PS^e execution and the goals succeed, then control returns to the C++ level. By posting a goal that fails, the ECL^{i}PS^e execution will fail back into the previous set of goals and these will succeed with a different solution.

```
#include        "eclipseclass.h"

main()
{
    ec_init();
    EC_ref Pred;

    post_goal(term(EC_functor("current_built_in",1),Pred));
    while (EC_succeed == EC_resume())
    {
        post_goal(term(EC_functor("writeln",1),Pred));
```

```
        post_goal(EC_atom("fail"));
    }
    ec_cleanup(0);
}
```

The above example prints all the built ins available in ECL^{i}PS^e. When `EC_resume()` returns `EC_succeed` there is a solution to a set of posted goals, and we print out the value of `Pred`. otherwise `EC_resume()` returns `EC_fail` to indicate that no more solutions to any set of posted goals is available.
It is possible also to cut such search. So for example one could modify the example above to only print the 10th answer. Initially one simply fails, but at the tenth solution one cuts further choices. Then one prints the value of 'Pred'.

```
#include        "eclipseclass.h"

main()
{
    ec_init();
    EC_ref Pred,Choice;
    int i = 0;

    post_goal(term(EC_functor("current_built_in",1),Pred));
    while (EC_succeed == EC_resume(Choice))
    {
        if (i++ == 10)
        {
            Choice.cut_to();
            break;
        }
        post_goal(term(EC_atom("fail")));
    }
    post_goal(term(EC_functor("writeln",1),Pred));
    EC_resume():
    ec_cleanup(0);
}
```

When `EC_resume()` is called with an `EC_ref` argument, this is for data returned by the `EC_resume()` If the return code is `EC_succeed` The `EC_ref` is set to a choicepoint identifier which can be used for cutting further choices at that point.

2.3.2 Asynchronous events

The posting of goals and building of any ECL^{i}PS^e terms in general cannot be done asynchronously to the ECL^{i}PS^e execution. It has to be done after the `EC_resume()` function has returned.
Sometimes it may be necessary to signal some asynchronous event to ECL^{i}PS^e, for example to implement a time-out. To do this one posts a named event to ECL^{i}PS^e. At the next synchronous point within the eclipse execution, the handler for that event is invoked.

```
/* C++ code, possibly within a signal handler */
post_event(EC_atom("timeout"));

/* ECLiPSe code */
handle_timeout(timeout) :-
    <appropriate action>

:- set_event_handler(timeout, handle_timeout/1).
```

2.3.3 The yield-resume model

Although implicitly yielding control when a set of goals succeeds or fails is often enough, it is possible to explicitly yield control to the C++ level. This is done with the **yield/2** predicate. This yields control to the calling C++ program. The arguments are used for passing data to C++ and from C++.
When **yield/2** is called within ECL^{i}PS^e code, the `EC_resume()` function returns the value `EC_yield` so one can recognise this case. The data passed out via the first argument of **yield/2** can be accessed from C++ via the `EC_ref` argument to `EC_resume()`. The data received in the second argument of **yield/2** is either the list of posted goals, or an `EC_word` passed as an input argument to `EC_resume()`.

```
yield(out(1,2),InData),
```

In this example the compound term `out(1,2)` is passed to C++. If we had previously called:

```
EC_ref FromEclipse;
result = EC_resume(FromEclipse);
```

then `result` would be `EC_yield` and `FromEclipse` would refer to `out(1,2)`. If then we resumed execution with:

```
result = EC_resume(EC_atom("ok"),FromEclipse);
```

then the variable `InData` on the ECL^{i}PS^e side would be set to the atom 'ok'.

2.3.4 Summary of EC_resume() arguments

`EC_resume()` can be called with two optional arguments. An input argument that is an `EC_word` and an output that is an `EC_ref`.
If the input argument is omitted, input is taken as the list of posted goals. Otherwise the input to ECL^{i}PS^e is exactly that argument.
If the output argument is present, its content depends on the value returned by `EC_resume()`. If it returns `EC_succeed` it is the choicepoint identifier. If it returns `EC_yield` It is the first argument to the **yield/2** goal. If it returns `EC_fail` it is not modified.

Chapter 3

Managing Data and Memory in Mixed-Language Applications

ECL^{i}PS^e is a software engine for constraint propagation and search tasks. As such, it represents its data in a form that is different from how it would be represented in a traditional C/C++ program. In particular, the ECL^{i}PS^e data representation supports automatic memory management and garbage collection, modifications that can be undone in a search context, referential transparency and dynamic typing.
In a mixed-language application, there are two basic ways of communicating information between the components coded in the different languages:

Conversion: When data is needed for processing in another language, it can be converted to the corresponding representation. This technique is appropriate for simple data types (integers, strings), but can have a lot of overhead for complex structures.

Sharing: The bulk of the data is left in its original representation, referred to by a handle, and interface functions (or methods) provide access to its components when required.

Both techniques are supported by the ECL^{i}PS^e/C and ECL^{i}PS^e/C++ interface.

3.1 Constructing ECL^{i}PS^e data

3.1.1 ECL^{i}PS^e atoms and functors

```
/* ECLiPSe code */
S = book("Gulliver's Tales","Swift",hardback,fiction),
```

In the above structure 'hardback' and 'fiction' are atoms. 'book' is the functor of that structure, and it has an arity (number of arguments) of 4.
Each functor and atom is entered into a dictionary, and is always referred to by its dictionary entry. Two classes, `EC_atom` and `EC_functor` are used to access such dictionary entries.
The 'Name' method applies to both, to get their string form. The 'Arity' method can be used to find out how many arguments a functor has.

```
/* C++ code */
EC_functor book("book",4);
```

```
EC_atom hardback("hardback");

if (book.Arity()) == 4) .. /* evaluates to true */
if (book == hardback) ..   /* evaluates to false */
s = hardback.Name();       /* like s = "hardback"; */
```

3.1.2 Building ECL^{i}PS^e terms

The `pword` C data type is used to store ECL^{i}PS^e terms. In C++ the `EC_word` data type is used. This is used for any C type as well as for ECL^{i}PS^e structures and lists. The size remains fixed in all cases, since large terms are constructed on the ECL^{i}PS^e global stack.
The consequences of this are that terms may be garbage collected or moved in memory whenever the ECL^{i}PS^e engine runs. This means that any C variables of type `pword` or any C++ variable of type `EC_word` must be considered invalid after every invocation of `EC_resume()`. Reusing an invalid `pword/EC_word` (for anything other than assigning a fresh value to it) may lead to a crash. To preserve a valid reference to a constructed term across invocations of `EC_resume()`, use the `ec_ref/EC_ref` term references described in section 3.3.

3.1.3 Building atomic ECL^{i}PS^e terms

It is possible to simply construct or cast from a number of simple C++ types to build an `EC_word` In addition, functions exist for creating new variables, and for the nil which terminates ECL^{i}PS^e lists.

```
/* making simple terms in C++ */
EC_word w;
EC_atom hardback("hardback");
w = EC_word("Swift");
w = EC_word(hardback);
w = EC_word(1.002e-7);
w = EC_word(12345);
w = EC_word(nil());
w = EC_word(newvar());

/* ECLiPSe equivalent code */
P1 = "Swift",
P2 = hardback,
P3 = 1.002e-7,
P4 = 12345,
P5 = [],
P6 = _,
```

3.1.4 Building ECL^{i}PS^e lists

The `list(head,tail)` function builds a list out of two terms. Well formed lists have lists as their tail term and a nil ("[]") at the end, or a variable at the end for difference lists.

```
/* making the list [1, "b", 3.0] in C++ */
EC_word w = list(1, list("b", list(3.0, nil())));
```

The following example shows how you can write functions to build variable length lists.

```
/* function to build a list [n,n+1,n+2,.....,m-1,m] */
EC_word fromto(int n, int m)
{
    EC_word tail = nil();
    for(int i = m ; i >= n ; i--)
        tail = list(i,tail);
    return tail;
}
```

The list is constructed starting from the end, so at all points during its construction you have a valid term. The interface is designed to make it hard to construct terms with uninitialised sub-terms, which is what you would need if you were to construct the list starting with the first elements.
Another way to construct a list is from an array of numbers:

```
/* making the list [11, 22, 33, 44, 55] in C++ */
long nums[5] = {11, 22, 33, 44, 55};
EC_word w = list(5, nums);
```

3.1.5 Building ECL^{i}PS^e structures

The `term(functor,args..)` function is used to build ECL^{i}PS^e structures. A number of different functions each with a different number of arguments is defined so as not to disable C++ casting which would be the case if we defined a function with variable arguments.

```
/* making s(1,2,3) in C++ */
EC_functor s_3("s",3);
EC_word w = term(s_3,1,2,3);
```

The above interface is convenient for terms with small fixed arities, for much larger terms an array based interface is provided.

```
/* making s(1,2,..,n-1,n) */
EC_word args[n];
for(int i=0 ; i<n ; i++)
    args[i] = i+1;
EC_word w = term(EC_functor("s",n),args);
```

3.2 Converting ECL^{i}PS^e data to C data

There are several aspects to examining the contents of a term. These include decomposing compound terms such as lists and structures, converting simple terms to C data types and testing the types of terms.
The functions for decomposing and converting check that the type is appropriate. If it is they return `EC_succeed` if not they return a negative error code.

3.2.1 Converting simple ECL^{i}PS^e terms to C data types

To convert from an ECL^{i}PS^e term to a C type you first have to declare a variable with that type. For fixed size data types (you can convert to `double`, `long` and `dident` fixed size data types) you are responsible for allocating the memory. For strings you declare a `char*` variable and on conversion it will point to the internal ECL^{i}PS^e string.
In the following example we see how one can try to convert to different types. Of course normally you will know what type you are expecting so only one of these functions need be called.

```
EC_word term;
double r;
long i;
EC_atom did;
char *s;
if (EC_succeed == term.is_double(&d))
    cout << d << "\n";
else if (EC_succeed == term.is_long(&i))
    cout << i << "\n";
else if (EC_succeed == term.is_atom(&did))
    cout << did.Name() << "\n";
else if (EC_succeed == term.is_string(&s))
    cout << s << "\n";
else
    cout << "not a simple type\n";
```

The term is converted by the function which returns `EC_success`. The functions that fail to convert will return a negative error number.
Care has to be taken with strings, these pointers point to the internal ECL^{i}PS^e string which may move or be garbage collected during an ECL^{i}PS^e execution. As a result if a string is to be kept permanently one should copy it first.

3.2.2 Decomposing ECL^{i}PS^e terms

The function `ec_get_arg(index,term,&subterm)` is used to get the index'th subterm of a structure. The index varies from 1 to arity of `term`. A list can also be decomposed this way, where the head is at index 1 and the tail at index 2.
Below we see how we would write a function to find the nth element of a list.

```
int nth(const int n,const EC_word list, EC_word& el)
{
    EC_word tail = list;
    for (int i=1, i<n, i++)
        if (EC_fail == tail.arg(2,tail))
            return EC_fail;
    return tail.arg(1,el);
}
```

The above function actually is not limited to lists but could work on any nested structure.

3.3 Referring to ECL^{i}PS^e terms

The terms constructed so far (as EC-words) have been volatile, that is they do not survive an ECL^{i}PS^e execution (due to eg. garbage collection), It is possible to create safe terms that have been registered with the ECL^{i}PS^e engine and which do survive execution. The `EC_ref` and `EC_refs` classes are provided for this purpose. `EC_refs` are vectors of safe terms.
When you declare an `EC_ref` it will contain free variables.

```
EC_ref X; /* declare one free variable */
EC_refs Tasks(10); /* declare 10 free variables */
```

`EC_ref`s work like logical variables. When ECL^{i}PS^e fails during search they are reset to old values. They are always guaranteed to refer to something i.e. they never contain dangling references. If ECL^{i}PS^e backtracks to a point in the execution older than the point at which the references were created, they return to being free variables, or take on their initial values.
It is possible to declare references, giving them an initialiser but this must be an atomic type that fits into a single word. That restricts you to atoms, integers and nil.
You can freely assign between an `EC_ref` and a `EC_word`.
One point to take care of is that assigning such a variable is not like unification since assignment cannot fail. It just overwrites the old value. Assignment is very similar to the **setarg/3** built-in in the ECL^{i}PS^e language.

3.4 Passing generic C or C++ data to ECL^{i}PS^e

It is possible to include any C or C++ data in an ECL^{i}PS^e term. To do this it is wrapped into a handle to tell ECL^{i}PS^e that this is external data. You also have to supply a method table, which is a set of functions that are called when ECL^{i}PS^e wants to make common operations that it assumes can be done on any data (eg. comparing, printing).

3.4.1 Wrapping and unwrapping external data in an ECL^{i}PS^e term

To create an ECL^{i}PS^e wrapper for a C/C++ object, the function `handle()` is used. It takes a pointer to any C or C++ data, and a pointer to a suitable method table (`t_ext_type` structure) and creates an ECL^{i}PS^e handle term which refers to them. Method tables for the common case of arrays of char, long or double are predefined. For example a handle for a double array is made like this

```
double my_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
EC_word w = handle(&ec_xt_double_arr, my_array);
```

where `ec_xt_double_arr` is a predefined method table for arrays of doubles. To convert back from an ECL^{i}PS^e term `is_handle()` is used. The method table passed in indicates the sort of data you expect to get. If the ECL^{i}PS^e handle contains the wrong sort, the function returns TYPE_ERROR:

```
if ((EC_succeed == w.is_handle(&ec_xt_double_arr, &obj))
    obj->my_method();
else
    cerr << "unexpected type\n";
```

The creation of the handle copies the address of the array, rather than the array itself, so the handle must be used within the scope of the array.

3.4.2 The method table

Apart from the predefined method tables ec_xt_double_arr, ec_xt_long_arr and ec_xt_char_arr, new ones can easily be defined. The address of the table is used as a type identifier, so when you get an external object back from ECL^{i}PS^e you can check its type to determine the kinds of operations you can do on it. You can choose not to implement one or more of these functions, by leaving a null pointer (`(void*)0`) in its field.

```
typedef void *t_ext_ptr;

typedef struct {
    void       (*free)       (t_ext_ptr obj);
    t_ext_ptr  (*copy)       (t_ext_ptr obj);
    void       (*mark_dids)  (t_ext_ptr obj);
    int        (*string_size)(t_ext_ptr obj, int quoted);
    int        (*to_string)  (t_ext_ptr obj, char *buf, int quoted);
    int        (*equal)      (t_ext_ptr obj1, t_ext_ptr obj2);
    t_ext_ptr  (*remote_copy)(t_ext_ptr obj);
    EC_word    (*get)        (t_ext_ptr obj, int idx);
    int        (*set)        (t_ext_ptr obj, int idx, EC_word data);
} t_ext_type;
```

void free(t_ext_ptr obj) This is called by ECL^{i}PS^e if it loses a reference to the external data. This could happen if the ECL^{i}PS^e execution were to fail to a point before the external object was created, or if a permanent copy was explicitly removed with built-ins like **setval/2**, **erase/2** or **bag_dissolve/2**. Note that an invocation of this function only means that *one* reference has been deleted (while the copy function indicates that a reference is added).

t_ext_ptr copy(t_ext_ptr obj) This is called by ECL^{i}PS^e when it wants to make a copy of an object. This happens when calling ECL^{i}PS^e built-ins like **setval/2** or **recordz/2** which make permanent copies of data. The return value is the copy. If no copy-method is specified, these operations will not be possible with terms that contain an object of this type. A possible implementation is to return a pointer to the original and e.g. increment a reference counter (and decrement the counter in the free-method correspondingly).

void mark_dids(t_ext_ptr obj) This is called during dictionary garbage collection. If an external object contains references to the dictionary (`dident`) then it needs to mark these as referenced.

int string_size(t_ext_ptr obj, int quoted)

int to_string(t_ext_ptr obj,char *buf, int quoted) When ECL^{i}PS^e wants to print an external object it calls `string_size()` to get an estimate of how large the string would be that represents it. This is used by ECL^{i}PS^e to allocate a buffer. The string representation must be guaranteed to fit in the buffer.

Finally the `to_string()` function is called. This should write the string representation of the object into the buffer, and return the actual size of that string.

int equal(t_ext_ptr obj1, t_ext_ptr obj2) This is called when two external objects are unified or compared. Prolog views the external object as a ground, atomic element. It should return non-zero if the objects are considered equal.

t_ext_ptr remote_copy(t_ext_ptr obj) This is called by parallel ECL^{i}PS^e when it needs to make a copy of an object in another worker. If the object is in shared memory, this method can be the same as the copy method.

EC_Word get(t_ext_ptr obj, int idx) Returns the value of a field of the C/C++ object. This methods gets invoked when the ECL^{i}PS^e predicate **xget/3** is called. The get/set method pair determines the mapping of index values to fields.

int set(t_ext_ptr obj, int idx, EC_word data) Set the value of a field of the C/C++ object. This methods gets invoked when the ECL^{i}PS^e predicate **xset/3** is called. The get/set method pair determines the mapping of index values to fields.

Example of the simplest possible user-defined method table:

```
/* the initializer is actually not needed, NULLs are the default */
t_ext_type my_type = {NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL};
my_struct data_in;
...
// creating a handle for data_in
EC_word w = handle(&my_type, &data_in);
...
// checking a handle and extracting the data pointer
my_struct *data_out;
if ((EC_succeed == w.is_handle(&my_type, &data_out))
    data_out->my_method();
else
    cerr << "unexpected type\n";
```

Chapter 4

External Predicates in C and C++

4.1 Coding External Predicates

External Predicates are C/C++ functions that can be called like predicates from ECL^{i}PS^e. Two following extra interface functions are provided for this purpose:

EC_word EC_arg(int i) returns the i'th argument of the predicate call.

pword ec_arg(int i)
same for C.

int unify(EC_word, EC_word)
unify two pwords. The return code indicates success or failure. Note however, that if attributed variables are involved, their handlers have not been invoked yet (this happens after the external predicate returns).

int EC_word::unify(EC_word)
same as method.

int ec_unify(pword, pword)
same for C.

Apart from that, all functions for constructing, testing and decomposing ECL^{i}PS^e data can be used in writing the external predicate (see chapter 3). Here are two examples working with lists, the first one constructing a list in C:

```
#include "eclipse.h"
int
p_string_to_list()         /* string_to_list(+String, -List) */
{
    pword  list;
    char *s;
    long len;
    int res;

    res = ec_get_string_length(ec_arg(1), &s, &len);
    if (res != PSUCCEED) return res;
```

```
    list = ec_nil();    /* build the list backwards */
    while (len--)
        list = ec_list(ec_long(s[len]), list);

    return ec_unify(ec_arg(2), list);
}
```

The next example uses an input list of integers and sums up the numbers. It is written in C++:

```
#include "eclipseclass.h"
extern "C" int
p_sumlist()
{
    int res;
    long x, sum = 0;
    EC_word list(EC_arg(1));
    EC_word car,cdr;

    for ( ; list.is_list(car,cdr) == EC_succeed; list = cdr)
    {
        res = car.is_long(&x);
        if (res != EC_succeed) return res;
        sum += x;
    }
    res = list.is_nil();
    if (res != EC_succeed) return res;
    return unify(EC_arg(2), EC_word(sum));
}
```

The source code of these examples can be found in directory doc/examples within the ECL^{i}PS^e installation.

4.2 Compiling and loading

It is strongly recommended to copy the makefile "Makefile.external" provided in your installation directory under lib/$ARCH and adapt it for your purposes. If the makefile is not used, the command to compile a C source with ECL^{i}PS^e library calls looks something like this:

```
% cc -G -I/usr/local/eclipse/include/sparc_sunos5
            -o eg_externals.so eg_externals.c
```

or

```
% cc -shared -I/usr/local/eclipse/include/i386_linux
            -o eg_externals.so eg_externals.c
```

If the external is to be used in a standalone ECL^{i}PS^e, it is possible to dynamically load it using the **load/1** predicate:

```
load("eg_externals.so")
```

On older UNIX platforms without dynamic loading, the following method may work. Compile the source using

```
% cc -c -I/usr/local/eclipse/include/sparc_sunos5 eg_externals.c
```

and load it with a command like

```
load("eg_externals.o -lg -lm")
```

The details may vary depending on what compiler and operating system you use. Refer to the Makefile.external for details.
Once the object file containing the C function has been loaded into ECL^{i}PS^e, the link between the function and a predicate name is made with **external/2**

```
external(sumlist/2, p_sumlist)
```

The new predicate can now be called like other predicates. Note that the **external/2** declaration must precede any call to the declared predicate, otherwise the ECL^{i}PS^e compiler will issue an *inconsistent redefinition* error. Alternatively, the **external/1** forward declaration can be used to prevent this.
If the external is needed in the context of an ECL^{i}PS^e which is itself embedded in a C/C++ host program, then the external code can be compiled/linked together with the host program, and the link between function and predicate name can alternatively be made by calling the C function ec_external(), e.g.

```
ec_external(ec_did("sumlist",2), p_sumlist, ec_did("eclipse"))
```

This must be done after the embedded ECL^{i}PS^e has been initialised (and after the module that is supposed to contain the external predicate has already been created).

4.3 Restrictions and Recommendations

It is neither supported nor recommended practice to call ec_resume() from within an external predicate, because this would invariably lead to programs which are hard to understand and to get right.
Currently, it is also not possible to post goals from within an external predicate, but that is a sensible programming style and will be supported in forthcoming releases. Posting events however is already possible now.

Chapter 5

Embedding into Tcl/Tk

This chapter describes how to embed ECL^{i}PS^e into a Tcl host program. Tcl/Tk is a cross-platform toolkit for the development of graphical user interfaces. The facilities described here make it possible to implement ECL^{i}PS^e applications with platform-independent graphical user interfaces. The interface is similar in spirit to the ECL^{i}PS^e embedding interfaces for other languages.
An alternative method of using ECL^{i}PS^e with Tcl is to use the Tcl remote interface, described in chapter 6. In this case, the ECL^{i}PS^e is ran as a separate program. The facilities provided by the remote and embedding interfaces are largely compatible, so that it is possible to reuse the same Tcl and ECL^{i}PS^e code in both interface. The advantage of the embedding interface is that ECL^{i}PS^e is much more tightly coupled with the Tcl program, and communication between the two is more efficient. The advantage of the remote interface is that the Tcl and ECL^{i}PS^e programs are not tightly coupled, and in fact can be run on separate machines.
The **tkeclipse** development environment is entirely implemented using the facilities described in this chapter. The toplevel of **tkeclipse** is currently implemented using only the embedding interface, but the development tools can be used with both the embedding and remote interfaces.

5.1 Loading the interface

The ECL^{i}PS^e interface is provided as a Tcl-package called **eclipse**, and can be loaded as follows:

```
lappend auto_path "/location/of/my/eclipse/lib_tcl"
package require eclipse
```

5.2 Initialising the ECL^{i}PS^e Subsystem

These are the Tcl commands needed to initialize an embedded ECL^{i}PS^e.

ec_set_option *option_name option_value*
: Set the value of an initialisation option for ECL^{i}PS^e. This must be done before invoking ec_init. The available option_names are: localsize, globalsize, privatesize, sharedsize, default_module, eclipsedir, io. See Appendix A for their meaning.

ec_init *?peername?*

Initialise the ECL^{i}PS^e engine. This is required before any other commands of this interface (except ec_set_option) can be used. The optional argument *peername* is the name of the embedding peer, which defaults to 'master'.

Example Tcl code for initialising ECL^{i}PS^e:

```
lappend auto_path "/location/of/my/eclipse/lib_tcl"
package require eclipse
#ec_set_option io 0;    # input/output/error via tty (for testing)
ec_set_option io 2;     # input/output/error via queues (default)
ec_init
```

Apart from the basic functionality in **package eclipse** which takes care of linking Tcl to ECL^{i}PS^e, there is a **package eclipse_tools** containing Tk interfaces to ECL^{i}PS^e facilities like debugging and development support. This package should be used when developing Tcl/Tk/ECL^{i}PS^e applications. To add these tools to your application, load the package and add the tools menu to your application's menu bar. Your code should then contain the following pattern:

```
package require eclipse
package require eclipse_tools
...
menu .mbar
...
ec_init
...
ec_tools_init .mbar.tools
```

See also the examples in the lib_tcl directory of the ECL^{i}PS^e installation.

5.3 Shutting down the ECL^{i}PS^e Subsystem

The embedded ECL^{i}PS^e is terminated when quitting from the Tcl/Tk application. The following Tcl command should be called just before the application is terminated to allow ECL^{i}PS^e to shutdown:

ec_cleanup

Shutdown the ECL^{i}PS^e engine.

5.4 Passing Goals and Control to ECL^{i}PS^e

The control flow between Tcl and ECL^{i}PS^e is conceptually thread-based. An ECL^{i}PS^e goal is executed by using the **ec_rpc** mechanism. The goal is posted from Tcl, and control is transferred automatically to ECL^{i}PS^e to allow the goal to be executed. Control can also be explicitly transferred to ECL^{i}PS^e using **ec_resume**. Furthermore, handler goals can be implicitly invoked on I/O operations on queues (this is described in more detail in section 5.5, with implicit transfer of control.
The related commands are the following:

ec_rpc *goal ?format?*

Remote ECL^{i}PS^e predicate call. It calls goal in the default module. The goal should be simple in the sense that it can only succeed, fail or throw. It must not call **yield/2**. Any choicepoints the goal leaves will be discarded.

Unlike **ec_resume**, calls to **ec_rpc** can be nested and can be used from within Tcl queue event handlers.

If no format argument is given, the goal is assumed to be in ECL^{i}PS^e syntax. If a *format* argument is provided, the ECL^{i}PS^e goal is constructed from *goal* and *format*, according to the conversion rules explained in section 5.8.

On success, **ec_rpc** returns the (possibly more instantiated) goal as a Tcl data structure, otherwise "fail" or "throw" respectively.

This is the recommended way of executing ECL^{i}PS^e code from Tcl, and passing the results back (via output arguments) to Tcl.

ec_running

checks whether an asynchronous ECL^{i}PS^e thread is still running. If that is the case, the only interface function that can be invoked reliably is **ec_post_event**.

ec_resume *?async?*

resume execution of the ECL^{i}PS^e engine: All posted events and goals will be executed. The return value will be "success" if the posted goals succeed, "fail" if the goals fail, and "yield" if control was transferred because of a **yield/2** predicate call in the ECL^{i}PS^e code. No parameters can be passed.

If the *async* parameter is 1 (default 0), the ECL^{i}PS^e execution is resumed in a separate thread, provided this is supported by the operating system. The effect of this is that Tcl/Tk events can still be handled while ECL^{i}PS^e is running, so the GUI does not freeze during computation. However, only one ECL^{i}PS^e thread can be running at any time, so before doing another call to **ec_resume**, **ec_handle_events** or **ec_rpc** one should use **ec_running** to check whether there is not a thread still running.

ec_flush *?stream_nr? ?nbytes?*

flushes the Tcl end of a to-ECL^{i}PS^e queue (see section 5.5) that has the ECL^{i}PS^e stream number *stream_nr*. Control is then briefly transferred to ECL^{i}PS^e so that any events that are raised can be handled. Afterwards the control is passed back to Tcl. *nbytes* is a dummy argument and is provided for compatibility with the Tcl remote interface only.

5.5 Communication via Queues

The most flexible way of passing data between ECL^{i}PS^e and Tcl is via the I/O facilities of the two languages, ie. via ECL^{i}PS^e queue streams which can be connected to Tcl channels.
Currently, a communication channel between ECL^{i}PS^e and Tcl is created from Tcl, which appears as an ECL^{i}PS^e queue in ECL^{i}PS^e, and a channel in Tcl. The queue has a symbolic name and a stream number in ECL^{i}PS^e, and has a channel name in Tcl. Facilities are provided to interconvert between these names.
Queues pass data between ECL^{i}PS^e and Tcl in one direction: either from ECL^{i}PS^e to Tcl (from-ECL^{i}PS^e), or from Tcl to ECL^{i}PS^e (to-ECL^{i}PS^e). Queues are created with the direction

specified. The queues should be viewed as communication channels: data is written to the queue, and it only becomes available to the other side when the queue is flushed. This is done by calling the predicate **flush/1** on the ECL^{i}PS^e side, and by invoking **ec_flush** on the Tcl side. The flush also has the effect of briefly transferring control to the other side to allow handlers to handle the data (see section 5.6)[1]

ec_queue_create *eclipse_stream_name mode ?command? ?event?*

Creates a queue between Tcl and ECL^{i}PS^e. On the Tcl side, a Tcl channel is created. On the ECL^{i}PS^e side, the queue would be given the symbolic name *eclipse_stream_name*. The *mode* argument indicates the direction of the queue, and can either be fromec or toec[2]. The procedure returns a channel identifier for use in commands like **puts**, **read**, **ec_read_exdr**, **ec_write_exdr** or **close**. The optional arguments *command* and *event* specifies the data handler for the queue: *command* is the name of the Tcl procedure for handling the data, with its user defined arguments. *event* is the name of the event that will be raised on the ECL^{i}PS^e side. As a handler can only be defined for one side, either *event* or *command* should be undefined (`{}`).

ec_queue_close *eclipse_stream_name*

Closes the queue with the ECL^{i}PS^e name of *ec_stream_name*.

ec_stream_nr *eclipse_stream_name*

This command returns the ECL^{i}PS^e stream number given a symbolic stream name (this is the same operation that the ECL^{i}PS^e built-in **get_stream/2** performs).

ec_streamname_to_streamnum *eclipse_stream_name*

This is an alias for *ec_stream_nr* for compatibility purposes.

ec_streamname_to_channel *eclipse_stream_name*

Returns the Tcl channel name for the queue with the symbolic name *eclipse_name*.

ec_streamnum_to_channel *eclipse_stream_number*

Returns the Tcl channel name for the queue with the ECL^{i}PS^e stream number *eclipse_stream_number*.

ec_async_queue_create *eclipse_stream_name mode ?command? ?event?*

This is provided mainly for compatibility with the Tcl remote interface. The command is an alias for **ec_queue_create** in the embedding interface. Certain uses of the queues in the embedding interface cannot be duplicated using the synchronous queues of the remote interface. Instead, asynchronous queues are needed (see chapter 6 for more details). This command is provided to allow the same code to be used for both interfaces. Note that it is possible to use the asynchronous queues of the remote interface in ways that are incompatible with the embedding interface.

[1] Strictly speaking, flushing is not necessary in the embedding case to make the data available to the other side. However, it is needed in the remote case, and for compatibility and good practice, flushing is recommended.

[2] For compatibility with previous versions of the embedding Tcl interface, the mode can also be specified as r (equivalent to fromec) or w (equivalent to toec). These can be somewhat confusing as read/write status depends on from which side the queue is viewed (a read queue in ECL^{i}PS^e is a write queue in Tcl).

5.5.1 From-ECL^{i}PS^e to Tcl

To create a queue from ECL^{i}PS^e to Tcl, use **ec_queue_create** with the *mode* argument set to **fromec**, e.g.

```
Tcl:            set my_in_channel [ec_queue_create my_out_queue fromec]
```

Once the queue is created, it can be used, e.g. by writing into it with ECL^{i}PS^e's **write/2** builtin, and reading using Tcl's **read** command:

```
ECLiPSe:        write(my_out_queue, hello),
                flush(my_out_queue).

Tcl:            set result [read $my_in_channel 5]
```

The disadvantage of using these low-level primitives is that for reading one must know exactly how many bytes to read. It is therefore recommended to use the EXDR (ECL^{i}PS^e external data representation, see section 5.8) format for communication. This allows to send and receive structured and typed data. The primitives to do that are **write_exdr/2** on the ECL^{i}PS^e side and ec_read_exdr (section 5.8) on the Tcl side:

```
ECLiPSe:        write_exdr(my_out_queue, foo(bar,3)),
                flush(my_out_queue).

Tcl:            set result [ec_read_exdr $my_in_channel]
```

In the example, the Tcl result will be the list `{foo bar 3}`. For details about the mapping see section 5.8.

5.5.2 To-ECL^{i}PS^e from Tcl

To create a queue from Tcl to ECL^{i}PS^e to Tcl, use **ec_queue_create** with the *mode* argument set to **toec**, e.g.

```
Tcl:            set my_out_channel [ec_queue_create my_in_queue toec]
```

Now the queue can be used, e.g. by writing into it with Tcl's **puts** command and by reading using ECL^{i}PS^e's **read_string/4** builtin:

```
Tcl:            puts $my_out_channel hello
                ec_flush [ec_streamname_to_streamnum my_in_queue] 5

ECLiPSe:        read_string(my_in_queue, "", 5, Result).
```

The disadvantage of using these low-level primitives is that for reading one must know exactly how many bytes to read, or define a delimiter character. It is therefore recommended to use the EXDR (ECL^{i}PS^e external data representation, see section 5.8) format for communication. This allows to send and receive structured and typed data. The primitives to do that are ec_read_exdr (section 5.8) on the Tcl side and **read_exdr/2** on the ECL^{i}PS^e side:

```
Tcl:            ec_write_exdr $my_out_channel {foo bar 3} (SI)
                ec_flush [ec_streamname_to_streamnum my_in_queue]

ECLiPSe:        read_exdr(my_in_queue, Result).
```

In the example, the ECL^{i}PS^e result will be the term `foo("bar",3)`. For details about the mapping see section 5.8.

5.6 Attaching Handlers to Queues

In order to handle ECL^{i}PS^e I/O on queues more conveniently, it is possible to associate a handler with every queue, either on the Tcl or ECL^{i}PS^e side. These handlers can be invoked automatically whenever the other side initiates an I/O operation.

5.6.1 Tcl handlers

To-ECL^{i}PS^e queues

For this purpose, the to-ECL^{i}PS^e queue must be created with the *command* argument set. The following example creates a queue that can be written from the ECL^{i}PS^e side, and whose contents, if flushed, is automatically displayed in a text widget:

```
Tcl:      pack [text .tout]
          ec_queue_create my_out_queue toec {ec_stream_to_window "" .tout} {}
```

Assume that ECL^{i}PS^e is then resumed, writes to the queue and flushes it. This will briefly pass control back to Tcl, the **ec_stream_to_window**-handler will be executed (with the stream number added to its arguments), afterwards control is passed back to ECL^{i}PS^e. Note that **ec_stream_to_window** is a predefined handler which reads the whole queue contents and displays it in the given text widget.

From-ECL^{i}PS^e queues

Similarly, a from-ECL^{i}PS^e queue could be created as follows:

```
Tcl:      ec_queue_connect my_in_queue fromec \
                      {ec_stream_input_popup "Type here:"} {}
```

Assume that ECL^{i}PS^e is then resumed and reads from my_in_queue. This will briefly yield control back to Tcl, the **ec_stream_input_popup**-handler will be executed, afterwards control is passed back to ECL^{i}PS^e.
Three predefined handlers are provided:

ec_stream_to_window *tag text_widget stream_nr*
: Inserts all the current contents of the specified queue stream at the end of the existing text_widget, using tag.

ec_stream_to_window_sync *tag text_widget stream_nr ?length?*
: This is provided for compatibility with the Tcl remote interface. This command is essentially an alias for *ec_stream_to_window*, with an optional dummy argument *length* that is ignored.

ec_stream_output_popup *label_text stream_nr*
Pops up a window displaying the label_text, a text field displaying the contents of the specified queue stream, and an ok-button for closing.

ec_stream_input_popup *label_text stream_nr*
Pops up a window displaying the label_text, an input field and an ok-button. The text typed into the input field will be written into the specified queue stream.

When ECL^{i}PS^e is initialised with the default options, its **output** and **error** streams are queues and have the **ec_stream_output_popup** handler associated. Similarly, the **input** stream is a queue with the **ec_stream_input_popup** handler attached. These handler settings may be changed by the user's Tcl code.

5.6.2 ECL^{i}PS^e handlers

A to-ECL^{i}PS^e queue can be configured to raise an ECL^{i}PS^e-event (see **event/1** and the User Manual chapter on event handling) whenever the Tcl program writes something into the previously empty queue. To allow that, the queue must have been created with the *event* argument of **ec_queue_create** set, e.g.[3]

```
Tcl:  set my_out_channel [ec_queue_create my_queue toec {} my_queue_event]
```

Assuming something was written into the queue from within Tcl code, the ECL^{i}PS^e event will be handled if the queue is flushed on the Tcl side with the command **ec_flush**:

```
Tcl:  puts $my_out_channel hello
      ec_flush [ec_streamname_to_streamnum myqueue]
```

If myqueue was empty, then the **puts** would raise the event `my_queue_event`. The **ec_flush** transfer control over to ECL^{i}PS^e, so that the event can be handled.

5.7 Obtaining the Interface Type

Generally, the Tcl embedded and remote interfaces are designed to allow the user to write code that can be used via both interfaces. However, sometimes it may be necessary to distinguish the two. This can be done via:

ec_interface_type
returns embedded for the embedding interface, and remote for the remote interface.

5.8 Type conversion between Tcl and ECL^{i}PS^e

EXDR (ECL^{i}PS^e External Data Representation, see also chapter 9) is a data encoding that allows to represent a significant subset of the ECL^{i}PS^e data types. The following Tcl primitives are provided to handle EXDR-format:

[3]It is possible to use the same name for both the queue stream itself and the event. This simplifies the event handler code because it receives that name as an argument.

ec_write_exdr *channel data ?format?*

write an EXDR-term onto the given channel. The term is constructed using the *data* argument and the additional type information provided in the *format* argument. If no format is specified, it defaults to S (string).

ec_read_exdr *channel*

reads an EXDR-term from the given channel and returns it as a Tcl data structure, according to its type. Note that, since Tcl does not have a strong type system, some typing information can get lost in this process (e.g. string vs. atom).

ec_tcl2exdr *data ?format?*

This is the low-level primitive to encode the given *data* and type information in *format* to an EXDR-string which is suitable for sending over communication links to ECL^{i}PS^e or other agents which can decode EXDR-format. If no format is specified, it defaults to S (string).

ec_exdr2tcl *exdr_string*

This is the low-level primitive to decode an EXDR-string. It returns a Tcl data structure, according to the type information encoded in the EXDR-string. Note that, since Tcl does not have a strong type system, some typing information can get lost in this process (e.g. string vs. atom).

Since Tcl is an untyped language, all commands which create EXDR terms accept, in addition to the data, an optional **format** argument which allows all EXDR data types to be specified. The syntax is as follows:

To create EXDR type	**use <format>**	**data required**
String	S	string (binary)
String	U	string (utf8)
Integer/Long	I	integer
Double	D	double
List	[<**formats**>]	fixed length list
List	[<**formats**>*]	list
Struct	(<**formats**>)	fixed list, first elem functor name
Struct	(<**formats**>*)	list, first elem functor name
Anonymous Variable	_	string "_"

Here are some examples that show which Tcl data/format pair corresponds to which ECL^{i}PS^e term (the curly brackets are just Tcl quotes and not part of the format string):

```
Tcl data        Tcl format      Eclipse term

hello           S               "hello"
hello           ()              hello
123             S               "123"
123             I               123
123             D               123.0
123             ()              '123'
{a 3 4.5}       {[SID]}         ["a", 3, 4.5]
```

```
{a 3 4.5}       S               "a 3 4.5"
{1 2 3 4}       {[I*]}          [1, 2, 3, 4]
{f 1 2 3}       {(I*)}          f(1,2,3)
{is _ {- 1 2}}  {(_(II))}       _ is 1-2
```

5.9 Incompatible and obsolete commands

Here is a list of commands in the embedding interface that are retained for compatibility purposes with previous versions. They have no equivalent in the Tcl remote interface, and their use for new code is discouraged.

ec_post_goal *goal ?format?*

post a goal that will be executed when ECL^{i}PS^e is resumed. If no *format* argument is given, the goal is taken to be a string in ECL^{i}PS^e syntax. Note that (unlike with the C/C++ interface) it is not possible to retrieve any variable bindings from ECL^{i}PS^e after successful execution of the goal. To pass information from ECL^{i}PS^e to Tcl, use queue streams as described later on. Example:

```
ec_post_goal {go("hello",27)}
```

If a *format* argument is provided, the ECL^{i}PS^e goal is constructed from *goal* data and *format*, according to the conversion rules explained in section 5.8. Example:

```
ec_post_goal {go hello 27} (SI)
```

Posting several goals is the same as posting the conjunction of these goals. Note that simple, deterministic goals can be executed independently of the posted goals using the **ec_rpc** command (see below).

ec_post_event *event_name*

Post an event to the ECL^{i}PS^e engine. This will lead to the execution of the corresponding event handler once the ECL^{i}PS^e execution is resumed. See also **event/1** and the User Manual chapter on event handling for more information. This mechanism is mainly recommended for asynchronous posting of events, e.g. from within signal handlers or to abort execution. Otherwise it is more convenient to raise an event by writing into an event-raising queue stream (see section 5.6.2).

ec_handle_events

resume execution of the ECL^{i}PS^e engine for the purpose of event handling only. All events that have been posted via ec_post_event or raised by writing into event-raising queues will be handled (in an unspecified order). The return value will always be "success", except when an asynchronous ECL^{i}PS^e thread is still running, in which case the return value is "running" and it is undefined whether the events may have been handled by that thread or not.

first create an ECL^{i}PS^e queue stream using ECL^{i}PS^e's **open/3** or **open/4** predicate, then connect that stream to a Tcl channel by invoking the **ec_queue_connect** command from within Tcl code.

ec_queue_connect *eclipse_stream_name mode ?command?*

Creates a Tcl channel and connects it to the given ECL^{i}PS^e stream (*eclipse_stream_name* can be a symbolic name or the ECL^{i}PS^e stream number). The *mode* argument is either r or w, indicating a read or write channel. The procedure returns a channel identifier for use in commands like **puts**, **read**, **ec_read_exdr**, **ec_write_exdr** or **close**. The channel identifier is of the form `ec_queueX`, where `X` is the ECL^{i}PS^e stream number of the queue. This identifier can either be stored in a variable or reconstructed using the Tcl expression

```
ec_queue[ec_stream_nr eclipse_stream_name]
```

If a *command* argument is provided, this command is set as the handler to be called when data needs to be flushed or read from the channel (see **ec_set_queue_handler**).

ec_set_queue_handler *eclipse_stream_name mode command*

Sets *command* as the Tcl-handler to be called when the specified queue needs to be serviced from the Tcl side. Unlike **ec_queue_connect**, this command does not create a Tcl channel. The *mode* argument is either r or w, indicating whether the Tcl end of the queue is readable or writable. For readable queues, the handler is invoked when the ECL^{i}PS^e side flushes the queue. The Tcl-handler is expected to read and empty the queue. For writable queues, the handler is invoked when the ECL^{i}PS^e side reads from the empty queue. The Tcl-handler is expected to write data into the queue. In any case, the handler *command* will be invoked with the ECL^{i}PS^e stream number appended as an extra argument.

Chapter 6

Remote Tcl Interface

This chapter describes the remote Tcl interface, which allows a separate external Tcl program to interact with ECL^{i}PS^e in much the same fashion as the embedding Tcl interface (see chapter 5). Like the embedding interface, Tcl and ECL^{i}PS^e code communicates by sending and receiving streams of bytes via I/O queues (the **ec_rpc** mechanism is implemented on top of these queues). The interface can thus be used in similar fashion to the embedding interface, e.g. for the development of graphical user interfaces to an ECLiPSe application, with the difference that the ECL^{i}PS^e program is a separate program and not embedded into the Tcl program.
The main features of the interface are:

- The connection between the Tcl and ECL^{i}PS^e processes are established via sockets using TCP network protocol. Thus the Tcl process can be run on a different machine and platform from the ECL^{i}PS^e process.
- The Tcl process can be attached to any running ECL^{i}PS^e process, including an ECL^{i}PS^e embedded into another host language.
- More than one Tcl (or other remote) process can be attached to a single ECL^{i}PS^e via the remote interface.
- For the programmer, the embedding and remote interfaces are largely similar, and once the connection is established in the remote interface, the same code on the ECL^{i}PS^e and Tcl sides can be used for both interfaces.

The remote interface thus offers more flexibility than an embedding interface in how the Tcl code can be connected to an ECL^{i}PS^e program. However, as the Tcl and ECL^{i}PS^e processes are not as tightly coupled as in an embedded interface, the speed of communications between the Tcl and ECL^{i}PS^e processes is likely to be slower.

6.1 Basic Concepts of the Interface

The interface is used by starting separate ECL^{i}PS^e and Tcl processes, and then **attaching** the Tcl process to the ECL^{i}PS^e process. Once attached, the Tcl and ECL^{i}PS^e processes can communicate much as in the embedded interface: ECL^{i}PS^e goals can be sent from the Tcl side to the ECL^{i}PS^e side via the remote predicate call (**ec_rpc**) mechanism, and further I/O queues can be established between the ECL^{i}PS^e and Tcl processes to allow streams of bytes to sent from one side to the other.

The attached Tcl process can also be detached from the ECL^{i}PS^e process. This disconnection will terminate and clean-up the links between the two processes. Thus, typically, if the programmer wants to allow a particular application to be usable through both the Tcl remote and embedding interfaces, the only code that needs to be specific to one or the other interface is the code associated with starting and termination of the application (the attach and detach operations in the case of the remote interface).
The interaction between the Tcl and ECL^{i}PS^e is mediated by a version of thread-like control flow of the embedded interface. The interface distinguishes two 'sides': the Tcl side, which is the Tcl process, and the ECL^{i}PS^e side, which is generally the ECL^{i}PS^e process[1]. At any given time, either the ECL^{i}PS^e side or the Tcl side has 'control'. When the Tcl side has control, execution of the ECL^{i}PS^e process is suspended. When the ECL^{i}PS^e side has control, the Tcl side cannot initiate the execution of ec_rpc goals. The interface can implicitly transfer control from one side to the other (e.g. when processing synchronous I/O), or it can be explicitly transferred.
An ECL^{i}PS^e process can have several attached remote processes. Each remote process is identified by a **control** name, which is the ECL^{i}PS^e name for a special control connection between the two sides.

6.2 Loading the Interface

Before using the interface, the Tcl program must first load a Tcl-package called **remote_eclipse**, which can be loaded as follows:

```
lappend auto_path "/location/of/my/eclipse/lib_tcl"
package require remote_eclipse
```

An ECL^{i}PS^e program, onto which the Tcl program will attach, also needs to be started.

6.3 Attaching and Initialising the Interface

To use the interface, the Tcl program needs to be attached to the ECL^{i}PS^e program. The attach request is initiated on the ECL^{i}PS^e side, by calling the predicate **remote_connect/3** [2] from ECL^{i}PS^e. The Tcl program is then attached to the ECL^{i}PS^e program by executing the procedure **ec_remote_init** from Tcl. If no error occurs, then the connection is established and the interface is set up.
In more detail, the ECL^{i}PS^e predicate remote_connect/3 establishes a socket listening for the connection from the Tcl side. It prints out, on the stream `log_output`, the hostname and the port number that the Tcl side should connect to:

```
[eclipse 1]: remote_connect(Host/Port, Control, _InitGoal).
Socket created at address chicken.icparc.ic.ac.uk/25909
```

On the Tcl side, ec_remote_init is called with the hostname and port number given by remote_connect/3:

[1] The ECL^{i}PS^e side may be more complicated than a simple ECL^{i}PS^e, as it can be an embedded ECL^{i}PS^e, or the ECL^{i}PS^e process and other attached remote processes.

[2] Instead of **remote_connect/3**, the more flexible **remote_connect_setup/3** and **remote_connect_accept/6** pair of predicates can be used. See the reference manual entries for these predicates for more details.

```
ec_remote_init chicken.icparc.ic.ac.uk 25909
```

ec_remote_init *host port ?init_command? ?pass? ?format?*

Initialise the remote Tcl interface on the Tcl side. A corresponding **remote_connect/3** must have been started on the ECL^{i}PS^e side, which specifies the hostname (*host*) and port number (*port*) to connect to. The optional *init_command* is an is a Tcl command that will be invoked at the end of the attachment, before control is handed over to the ECL^{i}PS^e side (see section 6.7 for more details). *pass* and *format* are optional arguments for a simple security check: they specify an ECL^{i}PS^e term that will be matched against a corresponding term (using **==/2**) on the ECL^{i}PS^e side before the connection is allow to proceed (*pass* will be sent to the ECL^{i}PS^e side in EXDR format[3]; the default is an empty string, which is what **remote_connect_setup/3** expects).

If successful, some initial links are established between the two sides, such as the control connection and the connection to allow rpc goals to be sent from the Tcl to the ECL^{i}PS^e side. After the attachment, optional user-defined initialisations are performed on both sides (via the InitGoal argument on the ECL^{i}PS^e side, and the init_command on the Tcl side), and the two sides can then interact. Initially, the control is given to the Tcl side, and *remote_connect/3* returns only when control is handed over to the ECL^{i}PS^e side.
As part of the attachment process, the ECL^{i}PS^e name of the control connection is passed to the Tcl side. This can be accessed by the user using the command:

ec_control_name

returns the ECL^{i}PS^e name of the control connection. An error is raised if this procedure is called before an attachment to ECL^{i}PS^e is made.

Unimplemented functionality error will be raised if the Tcl or ECL^{i}PS^e side are incompatible with each other. This can happen if one side is outdated, e.g. if the remote Tcl interface used and the ECL^{i}PS^e being connected to are not from the same version of ECL^{i}PS^e. In this case, it is best to update both sides to the latest version of ECL^{i}PS^e.

6.3.1 A Note on Security

Once a Tcl side is attached to an ECL^{i}PS^e, the Tcl side can execute ECL^{i}PS^e goals on the ECL^{i}PS^e side via the **ec_rpc** mechanism. This may be a security concern as this gives the Tcl side as much access to the resources on the ECL^{i}PS^e side as the ECL^{i}PS^e process itself, even though the Tcl side can potentially be anywhere reachable from the ECL^{i}PS^e side via TCP. However, the connection must be initiated from the ECL^{i}PS^e side, and the attachment process must follow a protocol in order for a successful attachment. Nevertheless, if a third party somehow knew which Address to connect to, and follows the protocol, it can 'steal' the connection to ECL^{i}PS^e. No authentication is performed by the simple `remote_connect_setup/3`, but **remote_connect_accept/6** does allow a simple authentication where it can require the Tcl side to send an ECL^{i}PS^e term that matches the one specified in calling the predicate. This is done before the Tcl side is given the ability to run rpc goals on the ECL^{i}PS^e side.
It is also possible to limit the remote connection to the same machine as the ECL^{i}PS^e process by specifying 'localhost' as the host name in the Host/Port address of `remote_connect/3`. The Tcl side must also use 'localhost' for the Host name in its client connection.

[3]See section 5.8 for more on EXDR format

Each peer queue is created by creating a new server socket on the ECL^{i}PS^e side and then accepting a client connection from the Tcl side. The accept command is told where the client connection is from, and the client host is checked against the client's host from the attachment, to ensure that the same host has been connected. If not, the ECL^{i}PS^e side will reject the particular connection. At this point, the security has probably been compromised, and the two side should disconnect.
Note also that by default, none of the information sent through the queues between the remote side and the ECL^{i}PS^e side is encrypted. If the programmer requires these communication channels to be secure, then such encryptions need to be provided by the programmer.

6.4 Type Conversion Between Tcl and ECL^{i}PS^e

The EXDR (ECL^{i}PS^e External Data Representation, see chapter 9) representation is fully supported by the interface. The same type conversions commands as in the embedding Tcl interface, described in section 5.8 (ec_write_exdr, ec_read_exdr, ec_tcl2exdr, ec_exdr2tcl), are available.

6.5 Executing an ECL^{i}PS^e Goal From Tcl

An ECL^{i}PS^e predicate can be invoked from the Tcl side using the remote ECL^{i}PS^e predicate call (ec_rpc) facility. This should be the main method of interacting and communicating with ECL^{i}PS^e in the remote interface. Information can be sent to ECL^{i}PS^e via bindings for (input) arguments when the call is made; and results returned from ECL^{i}PS^e via the bindings made to (output) arguments:

ec_rpc *goal ?format?*

Remote ECL^{i}PS^e predicate call. It calls goal in the default module. The goal should be simple in the sense that it can only succeed, fail or throw. Any choice-points the goal leaves will be discarded.

Calls to **ec_rpc** can be nested and can be used from within Tcl queue event handlers. However, an **ec_rpc** cannot be issued while ECL^{i}PS^e side has control.

If no format argument is given, the goal is assumed to be in ECL^{i}PS^e syntax. If a *format* argument is provided, the ECL^{i}PS^e goal is constructed from *goal* and *format*, according to the conversion rules explained in section 5.8.

On success, **ec_rpc** returns the (possibly more instantiated) goal as a Tcl data structure (in EXDR format), otherwise "fail" or "throw" respectively.

6.6 Communication via Queues

Queues should be used to set up long-term I/O links between ECL^{i}PS^e and Tcl. An example would be the main output from an application that is to be displayed by a Tcl window. Streams of bytes can be sent along the queue from one side to the other: on one side, data is written to the queue; and when the queue is flushed, the data is sent to the other side, which can now read the data. The data can either be sent as normal strings (where each byte represents a character) using the normal I/O calls, or they can be in EXDR format, in which case both sides need to read and write using EXDR.

On the Tcl side, the queue is seen as a Tcl I/O channel. On the ECL^{i}PS^e side, a queue is seen as an ECL^{i}PS^e I/O stream, which has a unique (numeric) ID, the stream number, and has a user supplied symbolic name. These all refer to the same queue. Queues are created using the symbolic names, and the Tcl side maintains tables of the correspondence between Tcl channel names, symbolic names and stream numbers. The built-in Tcl I/O commands accepts the Tcl channel name for operating on the queue, and for compatibility with the embedding interface, many of the Tcl remote interface commands refer to the queue using the stream number. The interface provides commands to inter-convert the various names so that the right name can be used for a particular command.
There are two types of queues:

synchronous These queues are unidirectional, i.e. either for sending data from ECL^{i}PS^e to Tcl (from-ECL^{i}PS^e), or from Tcl to ECL^{i}PS^e (to-ECL^{i}PS^e). These streams are synchronous because the interface ensures that the sending and receiving of data across the queue are synchronised. This is achieved by transferring control between ECL^{i}PS^e and Tcl in a coroutine-like manner to ensure that data that is sent from one side is processed on the other.

These queues are designed to be compatible with the queues created via ec_queue_create of the embedded interface (see section 5.5). Their actual implementations are different, in that the queues in the embedded case are memory queues and the synchronous queue use socket connections. The interface tries to minimise the difference by buffering where possible at either ends of the socket connection. However, there is an overhead for doing this, and not all differences can be hidden. This is discussed in more detail in section 6.9.

asynchronous These are bi-directional – data can be sent both ways. Sending or receiving data on these queues does not necessarily transfer control between ECL^{i}PS^e and Tcl. In particular, it is not possible to request data from the other side if the queue is empty: such an operation would simply block. This is because such queues map directly to the socket connections with no buffering, and there is no concept of a socket being empty. Generally, it is up to the programmer to co-ordinate the transfer and processing of the data.

They have no direct equivalent in the embedding Tcl interface, but some uses of the embedding Tcl interface queues, such as writing data from one side without a corresponding reader on the other side, are better approximated by the asynchronous queues than the synchronous queues. They can also be more efficient in that there is no buffering of the data is performed by the interface.

6.6.1 Queue Data Handlers

The processing of data on queues (synchronous and to some extent asynchronous) can be performed via *handlers*. A handler is a piece of code (a procedure in Tcl, a goal in ECL^{i}PS^e) whose execution is data-driven: it is invoked to deal with the transfer of data on a queue on their respective sides.
In ECL^{i}PS^e, the handler goal is invoked using the events mechanism. That is, an event is raised, and the event handler goal associated with the event (see **set_event_handler/2**) is then executed when ECL^{i}PS^e has control.
A handler can be called under two situations:

Data consumer To consume data that has been sent over from the other side. Here, the other side has sent data over the queue, invoking the handler. The handler is expected to read the data off the queue and process it. An example of a data consumer handler is a Tcl handler which is invoked when the ECL^{i}PS^e side sends data that is intended to be displayed on a Tcl window. The handler would be invoked to read the data off the queue and display it on the window.

Data provider To provide data that has been requested by the other side. In this case, the handler is expected to generate the data and write the data onto the queue, and send it to the other side. For example, on the Tcl side, a Tcl handler might be invoked to ask for inputs from the user via the GUI. Note that these data providers can only exist for the synchronous queues.

For each queue and for a particular direction of data flow, a handler can be defined on either the Tcl or the ECL^{i}PS^e side, but not both. The handler either consumes or provides data as described above. The reason that handlers cannot be defined on both sides is that this avoids possible infinite loop of alternately triggering the data provider and the data consumer.

6.6.2 Synchronous Queues

These queues can be created on the Tcl side. This is done with the **ec_queue_create** command from within Tcl code:

ec_queue_create *eclipse_stream_name mode ?command? ?event?*

Creates a synchronous queue between Tcl and ECL^{i}PS^e sides. On the Tcl side, a Tcl channel is created. On the ECL^{i}PS^e side, the queue would be given the symbolic name *eclipse_stream_name*. The *mode* argument indicates the direction of the queue, and can either be fromec or toec[4]. The procedure returns a channel identifier for use in commands like **puts**, **read**, **ec_read_exdr**, **ec_write_exdr** or **close**. The optional arguments *command* and *event* specifies the data handler for the queue: *command* is the name of the Tcl procedure for handling the data, with its user defined arguments. *event* is the name of the event that will be raised on the ECL^{i}PS^e side (see the section 6.6.1 for more details). As a handler can only be defined for one side, either *event* or *command* should be undefined (`{}`).

ec_queue_close *eclipse_stream_name*

Closes the (synchronous or asynchronous) queue with the ECL^{i}PS^e name of *ec_stream_name*. The queue is closed on both the Tcl and ECL^{i}PS^e sides, and book-keeping information for the queue is removed.

It is strongly recommended that the queues should be used for long-term I/O connections between the two sides, and so the queues should not be created and closed on a short-term basis. For quick interchange of data, it is recommended that the user use the **ec_rpc** mechanism.

[4]For compatibility with previous versions of the embedded Tcl interface, the mode can also be specified as r (equivalent to fromec) or w (equivalent to toec). These can be somewhat confusing as read/write status depends on from which side the queue is viewed (a read queue in ECL^{i}PS^e is a write queue in Tcl).

Handlers for a Synchronous From-ECL^{i}PS^e Queue

Tcl Handler for From-ECL^{i}PS^e Queue For a from-ECL^{i}PS^e queue, the Tcl handler *command* would be a data consumer. This handler is initiated when ECL^{i}PS^e side initially has control and flushes the queue (by calling **flush/1**). With a Tcl handler defined, control is transferred to the Tcl side, where *command* is invoked to consume the data. When the handler finishes, control is returned to the ECL^{i}PS^e side. The general sequence of actions are:

ECL^{i}PS^e side	Tcl side
Writes to the from-ECL^{i}PS^e queue	
Flush the from-ECL^{i}PS^e queue	
	Handler invoked to handle data on the from-ECL^{i}PS^e queue
ECL^{i}PS^e returns from flush, and continue executing the following code	

The Tcl handler is specified by *command* in **ec_queue_create**. *command* includes the name of the Tcl procedure to invoke, and any user defined arguments. When the handler is invoked, two additional arguments are appended: the ECL^{i}PS^e stream number for the queue, and the number of bytes that has been sent on the queue. This command should read the data off the queue and process it. The following predefined Tcl data consumer handlers are provided:

ec_stream_to_window_sync *tag text_widget stream_nr length*
: Read *length* bytes from the specified queue and insert the data at the end of the existing *text_widget*, using *tag* as the tag for the text. If this is invoked as a handler for a from-ECL^{i}PS^e queue, *length* and *stream_nr* would be supplied when the handler is invoked.

ec_stream_output_popup *label_text stream_nr length*
: Pops up a window displaying the *label_text*, a text field displaying the contents of the specified queue stream, and an ok-button for closing. The data is read as normal strings. This is the default Tcl fromec handler that will be called if **ec_create_queue** did not define one.

An example from-ECL^{i}PS^e queue with Tcl handler To create the queue on the Tcl side with a Tcl handler:

```
Tcl code :  ec_queue_create myqueue fromec {ec_stream_to_window_sync red textwin} {}
```

Note that the last `{}` specifies that there is no ECL^{i}PS^e handler. This is the actual default for this argument, so it could be missed out. After creating the queue, it can be used on the ECL^{i}PS^e side. The programmer can write to the queue, and to send the data to the Tcl side, the queue should be flushed:

```
ECLiPSe code :
   ...
   write(myqueue, hello),
   flush(myqueue),
   ...
```

When the queue is flushed as shown above, then control is handed over to Tcl, and the Tcl handler, in this case **ec_stream_to_window_sync**, would be invoked. This reads the data on the queue (hello, and anything else that has been written since the last flush), and puts it into the text widget textwin, with the tag red. The procedure is also called with the ECL^{i}PS^e stream number for the queue and the number of bytes sent as extra arguments. The textwin widget and the tag red must be defined already in the Tcl program (presumably 'red' means printing the text in red colour); if no tag is desired, `{}` can be used.
The procedure **ec_stream_to_window_sync** is predefined in the interface, but here is a slightly simplified version of it:

```
proc ec_stream_to_window_sync {Tag Window Stream Length} {

    set channel [ec_streamnum_to_channel $Stream]
    set data [read $channel $Length]

    $Window insert end $data $Tag
    $Window see end
}
```

ECL^{i}PS^e Handler for From-ECL^{i}PS^e Queue Currently, the Tcl remote interface does not support ECL^{i}PS^e handlers (which will be a data provider) for from-ECL^{i}PS^e queues. Thus, the *event* argument for **ec_queue_create** is currently a dummy argument that is ignored. The available alternative is to use **ec_rpc** to obtain the required information: instead of reading from a from-ECL^{i}PS^{e}queue, an ec_rpc should be called with argument(s) left to be filled in by the ECL^{i}PS^e side with the required data.

Handlers for a Synchronous To-ECL^{i}PS^e Queue

Tcl Handler for a To-ECL^{i}PS^e Queue For a to-ECL^{i}PS^e queue, the Tcl handler *command* defined in **ec_queue_create** would be a data producer. This handler is initiated when ECL^{i}PS^e side has control, and reads from the to-ECL^{i}PS^e queue, which is initially empty. With a Tcl handler defined, control is transferred to the Tcl side, where *command* is invoked to provide the data. The handler should write the data to the queue, and call the Tcl remote interface command **ec_flush** to send the data to ECL^{i}PS^e side. When the handler finishes, control is returned to the ECL^{i}PS^e side, and the read operation is performed to read the now available data. The general sequence of actions are:

ECL^{i}PS^e side	Tcl side
Reads an empty to-ECL^{i}PS^e queue	Handler invoked to supply data to the to-ECL^{i}PS^e queue. The data is written to the queue and flushed with **ec_flush**
ECL^{i}PS^e returns from the initial read operation, reading the data supplied by the Tcl handler, and continue execution the following code	

The Tcl remote interface command **ec_flush**, instead of the standard Tcl **flush** command, should be used to flush a queue so that the data would be transferred and processed on the ECL^{i}PS^e side. **ec_flush** should be used both inside the Tcl data provider handler, and also to invoke an ECL^{i}PS^e data consumer handler (see the next section).

ec_flush *eclipse_streamnum ?nbytes?*

> If the Tcl side has control, flushes the (synchronous or asynchronous) queue with the ECL^{i}PS^e stream number *eclipse_streamnum* and hands over control briefly to ECL^{i}PS^e to read the data. Control is then returned to Tcl. *nbyte* is an optional argument that specifies the number of bytes being sent. If this argument is missing, the data sent must be a single EXDR term in the case of the synchronous queue. There is no restriction for the asynchronous queues, but it is the programmer's responsibility that the read operation does not block.

Normally, data is written to the queue using standard Tcl output commands, and the amount of data written is not known. However, the programmer may have kept track of the number of bytes written inside the handler, and thus know how many bytes will be sent. In this case, **ec_flush** can be called with the number of bytes supplied as a parameter. It is the programmer's responsibility to ensure that this information is accurate. Without nbytes, the output is restricted to EXDR terms for synchronous queues. The reason for this is because the data is sent through a socket connection, and without knowing the amount of data, it is not possible in general to know when the data ends, unless the data sent has implicit boundaries, like an EXDR term.
For the use of **ec_flush** inside a Tcl data provider handler, the sequence of events that appears to the user is that the **ec_flush** flushes the data, and the Tcl side then continues executing Tcl code until the handler's execution is finished. Control is then returned to ECL^{i}PS^e, where the original read operation can now read the available data. The actual sequence of event is slightly more complex for synchronous queues: when **ec_flush** is invoked, control is actually transferred to ECL^{i}PS^e, and the data flushed is then read into a buffer by ECL^{i}PS^e, which then returns control to Tcl to continue the execution of the handler. When the handler finally finishes, control returns to ECL^{i}PS^e, and the original read operation reads the data from the buffer and continues. This extra complexity should be transparent to the programmer except when the intermediate ECL^{i}PS^e read to buffer does not complete (e.g. because *nbytes* is greater than the actual amount of data sent).
The Tcl handler is specified by *command* in **ec_queue_create**. *command* includes the name of the Tcl procedure to invoke, and any user defined arguments. When the handler is invoked, an additional argument is appended: the ECL^{i}PS^e stream number for the queue. This command should get the data required, output it onto the queue, and call **ec_flush** to flush the data to ECL^{i}PS^e side. If the command does not flush data to ECL^{i}PS^e, ECL^{i}PS^e will print a warning and return control to Tcl side.
The following predefined Tcl data producer handler is provided:

ec_stream_input_popup *label_text stream_nr*

> Pops up a window displaying the label_text, an input field and an ok-button. The text typed into the input field will be written into the specified queue stream *stream_nr*, which is the ECL^{i}PS^e stream number for the queue. If this command is invoked as a handler for a to-ECL^{i}PS^e queue, *stream_nr* will be automatically appended by the interface. There should be no unflushed data already on the queue when this command is invoked.

An example to-ECL^{i}PS^e queue with Tcl handler To create the queue on the Tcl side with a Tcl-handler:

```
Tcl code :
ec_queue_create myqueue toec \
    {ec_stream_input_popup "Input for myqueue:"} {}
```

This associates the pre-defined Tcl data producer handler **ec_input_popup** with myqueue. The last {} specifies that there is no ECL^{i}PS^e handler and can be omitted as that is the default. This queue can now be used on the ECL^{i}PS^e side in a demand driven way, i.e. ECL^{i}PS^e side can read from the queue:

```
ECLiPSe code :

    ...
    read(myqueue, Data),
    ...
```

When the ECL^{i}PS^e side reads from myqueue, and the queue contains no data on the ECL^{i}PS^e side, then control will be handed over to Tcl, and **ec_input_popup** invoked. This pops up a Tcl window, with the label "Input for myqueue:" with a text entry widget, asking the user to supply the requested data. The data is then sent back to the ECL^{i}PS^e side.
Here is a slightly simplified version (there are no buttons) of **ec_stream_input_popup**:

```
set ec_stream_input_string {}

proc ec_stream_input_popup {Msg Stream} {
    global ec_stream_input_string

    toplevel .ec_stream_input_box
    label .ec_stream_input_box.prompt  -width 40 -text $Msg
    entry .ec_stream_input_box.input -bg white -width 40 \
        -textvariable ec_stream_input_string
    bind .ec_stream_input_box.input <Return> {destroy .ec_stream_input_box}

    ;# pack the popup window
    pack .ec_stream_input_box.prompt -side top -fill x
    pack .ec_stream_input_box.input -side top -fill x

    tkwait window .ec_stream_input_box
    puts -nonewline [ec_streamnum_to_channel $Stream] $ec_stream_input_string
    ;# flush the output to ECLiPSe with the length of the input
    ec_flush $Stream [string length $ec_stream_input_string]
}
```

Data is flushed to the ECL^{i}PS^e side using **ec_flush**. The **puts** needs the Tcl channel name of the queue to write to, and this is provided via the Tcl remote interface command

ec_streamnum_to_channel (see section 6.6.5). **ec_flush** is called with two arguments in this case, both the queue number (*Stream*), and the length of the data that is sent. Note that this makes the assumption that no other unflushed data has been written to the queue.

ECL^{i}PS^e Handler for a To-ECL^{i}PS^e Queue For a to-ECL^{i}PS^e queue, the ECL^{i}PS^e handler would be a data consumer. This handler is initiated when Tcl initially has control, and flushes data on a queue using **ec_flush**. Control is transferred to ECL^{i}PS^e, and if the ECL^{i}PS^e handler is defined, this is invoked to consume the data. When the handler returns, control is returned to Tcl, which continues executing the code after the flush. The general sequence of actions are:

ECL^{i}PS^e side	Tcl side
	Outputs data onto the to-ECL^{i}PS^e queue
	Calls **ec_flush** to send data to ECL^{i}PS^e side
The ECL^{i}PS^e handler associated with the queue is called to consume and process the data	
	Execution continues after the **ec_flush**

The ECL^{i}PS^e handler is specified by the *event* argument of **ec_queue_create**. This specifies an event that will be raised on the ECL^{i}PS^e side when data is written to a previously empty queue. The ECL^{i}PS^e side does not see this data, and the event not raised, until the data is flushed by **ec_flush** and copied by ECL^{i}PS^e to its buffer and, if the buffer was initially empty, the event would then be raised.
The programmer should define the event handler associated with *event*.

An example to-ECL^{i}PS^e queue with ECL^{i}PS^e handler To create the queue on the Tcl side with an ECL^{i}PS^e-handler:

```
Tcl code:
    ec_queue_create myqueue toec {} remoteflush_myqueue
```

Note that the `{}` is needed to specify that there is no Tcl handler. It defines `remoteflush_myqueue` as the event that will be raised when the queue is flushed by **ec_flush** on the Tcl side.
The event handler needs to be defined on the ECL^{i}PS^e side:

```
ECLiPSe code:

:- set_event_handler(remoteflush_myqueue, read_myqueue/0).

...
read_myqueue :-
      read_exdr(myqueue, Data),
      process(Data).
```

This read handler assumes that the data is written using EXDR format. So on the Tcl side, the data should be written using EXDR format:

Tcl code:

```
...
ec_write_exdr [ec_streamname_to_channel myqueue] $data
ec_flush [ec_streamname_to_streamnum myqueue]
...
```

6.6.3 Asynchronous Queues

Asynchronous queues are created on the Tcl side using the Tcl command **ec_async_queue_create**:

ec_aysnc_queue_create *eclipse_stream_name ?mode? ?fromec_command? ?toec_event?*
: Creates a socket stream between ECL^{i}PS^e and Tcl with the name *eclipse_stream_name* on the ECL^{i}PS^e side. The created stream is bidirectional, and can be written to or read from at both ends. The *mode* argument is for compatibility with the *ec_aysnc_queue_create* command of the embedded interface only, and has no effect on the nature of the queue. The procedure returns a channel identifier for use in commands like **puts**, **read**, **ec_read_exdr**, **ec_write_exdr** or **close**. Unlike the synchronous queues, only data consumer handlers can be defined: if a *fromec_command* argument is provided, this command is set as the Tcl data consumer handler to be called when data arrives on the Tcl end of the socket. If *toec_event* is given, it specifies the event that will be raised on the ECL^{i}PS^e side when data is flushed by *ec_flush* on the Tcl side.

ec_queue_close *eclipse_stream_name*
: Closes the (synchronous or asynchronous) queue with the ECL^{i}PS^e name of *ec_stream_name*. The queue is closed on both the Tcl and ECL^{i}PS^e sides, and book-keeping information for the queue is removed.

Asynchronous queues are bi-directional queues which allows data transfer between ECL^{i}PS^e and Tcl sides without transfer of control. In the case where a Tcl data consumer handler is defined in *fromec_command*, which is invoked on the Tcl side when the queue is flushed on the ECL^{i}PS^e side, the ECL^{i}PS^e side will carry on execution while the handler is invoked on the Tcl side.
These queues are designed to allow for more efficient transfer of data between ECL^{i}PS^e and Tcl than the synchronous queues.
For data transfer from ECL^{i}PS^e to Tcl, the intended use is that a Tcl data consumer handler would be invoked as the data becomes available on the Tcl side, after being flushed from the ECL^{i}PS^e side. Note that control is not handed over to Tcl side in this case: the Tcl handler is invoked and executed on the Tcl side while ECL^{i}PS^e side still has control, with the restriction that the Tcl handler is unable to issue **ec_rpc** goals because ECL^{i}PS^e side still retains control. Another difference with the synchronous from-ECL^{i}PS^e queue is that the handler would read from the queue in non-blocking mode, i.e. it will read whatever data is available on the queue at the Tcl side and never wait for more data. If more data become available, the handler would be invoked again. The following Tcl handler is pre-defined for the asynchronous queue for handling from-ECL^{i}PS^e data:

ec_stream_to_window *tag text_widget stream_nr length*

Inserts all the current contents of the specified queue at the end of the existing *text_widget*, using *tag* as the tag for the text.

For data transfer from Tcl to ECL^{i}PS^e, the queue can be used either asynchronously or synchronously. If the queue is used asynchronously, then the standard Tcl command **flush** should be used to flush the queue. There would not be any transfer of control, and so there would not be an immediate corresponding read on the ECL^{i}PS^e side. In fact, no handler would be invoked automatically on the ECL^{i}PS^e side, even when control is transferred. Output and flush operations do not block on the Tcl side, as the Tcl side of the queue is put into non-blocking mode, so that the data is buffered and the operations carried out when they will not block. It is the programmer's responsibility to write and call the code to read the data from the queue on the ECL^{i}PS^e side when the ECL^{i}PS^e side is given control.
This asynchronous use to send data to ECL^{i}PS^e should be useful when the queue is used as an auxiliary data channel, where the main data is sent either via **ec_rpc** or another queue. The desired effect is that data can be sent on the auxiliary channel without triggering processing on the ECL^{i}PS^e side until it is told to do so on the main data channel, which would be handled synchronously.
To use the queue synchronously for to-ECL^{i}PS^e data, **ec_flush** should be used to flush the queue on the Tcl side. With **ec_flush**, control will be handed over to the ECL^{i}PS^e side to process the data: the goal associated with the event *toec_event* is executed, and this goal should read the data from the queue. Unlike the synchronous to-ECL^{i}PS^e queues, the data is not buffered, and the handler goal is called every time **ec_flush** is invoked, rather than only when the queue is empty. This should normally not make any difference, as the handler should empty all the contents of a queue each time it is invoked.
The goal is called with two optional arguments: the first argument is the event name, the second argument is the 'culprit' of the form `rem_flushio(Queue,Len)`, indicating that this event is caused by a remote flush, where Queue is the ECL^{i}PS^e stream number, and Len is the number of bytes sent (this is supplied by **ec_flush**, if **ec_flush** does not supply a length, then Len is the atom `unknown`.

Examples for asynchronous queue

Using the queue asynchronously: to-ECL^{i}PS^e An example of using an asynchronous queue asynchronously to send data to ECL^{i}PS^e is in the tracer for TkECL^{i}PS^e development tools. Here the trace line is printed on a synchronous from-ECL^{i}PS^e queue, and handled by a Tcl data consumer handler which prints the trace line and waits for the user to type in a debugger command. This debugger command is sent to the ECL^{i}PS^e-side using an asynchronous queue, which is read by the ECL^{i}PS^e side when it returns. Here is a much simplified version of the code:

```
Tcl code:
    ...
    ec_queue_create debug_traceline fromec handle_trace_line
    ec_async_queue_create debug_input toec ;# no handlers
    ...
```

During the initialisation of the development tools, the Tcl code creates the from-ECL^{i}PS^e queue where the trace-line information is sent (`debug_traceline`), and the asynchronous queue (used

only in a to-ECL^{i}PS^e direction) for sending the debugger commands to ECL^{i}PS^e (creep, leap, skip etc.). Note that as this queue is used asynchronously, there are no handlers associated with it.
On the ECL^{i}PS^e side, when a goal with a spy-point is executed, this raises an event that calls the predicate `trace_line_handler/2` which should output the trace-line, and wait for a debug command, process the command, and carry on:

```
trace_line_handler(_, Current) :-
        % Current contains information on the current execution state
        % from this a trace line Traceline (a string) can be created
        make_current_traceline(Current, Traceline),
        % send the traceline to Tcl side
        write_exdr(debug_traceline, Traceline),
        flush(debug_traceline),
        % flush will return when the Tcl handler has finished
        read_exdr(debug_input, Cmd),
        % read the command from debug_input and process it
        interpret_command(Cmd, Current).
```

The trace-line handler is called with the second argument set to a structure that contain information on the current execution state (`Current`), from this, a trace-line (the debug port name, depth, goal being traced etc.) can be constructed: `Traceline` is the string that should be printed, e.g.

```
  (1) 1 CALL  append([1, 2, 3], [], L)
```

This is sent as an EXDR term to the Tcl side using the synchronous queue `debug_traceline`. When `flush/1` is called, control is handed over to the Tcl to handle the data, and the Tcl data consumer handler **handle_trace_line** is invoked:

```
proc handle_trace_line {stream length} {
global tkecl

        $ec_tracer.trace.text insert end \
             [ec_read_exdr [ec_streamnum_to_channel $stream]]
        configure_tracer_buttons active

        ;# wait for a tracer command button to be pressed...
        tkwait variable tkecl(tracercommand)
configure_tracer_buttons disabled
        ec_write_exdr [ec_streamname_to_channel debug_input] \
             $tkecl(tracercommand)
        flush [ec_streamname_to_channel debug_input]
}
```

As this is invoked as a handler, the ECL^{i}PS^e stream number (*stream*) and number of bytes sent (*length*) are appended as arguments. Note that as the trace-line is written as an EXDR term, the *length* information is actually not needed. What the handler does is simply read

the trace-line as an EXDR term, and placing the resulting string onto the tracer text window `$ec_tracer_trace.text`. Next, **configure_tracer_buttons active** is called. This code is not shown, but what it does is to enable the buttons for the debugger commands so that the user can press them. There are buttons for the debugger commands such as 'leap', 'creep' etc. When one of this button is pressed, the global variable `tkecl(tracercommand)` is set to the corresponding command, and the handler continues its execution beyond the **tkwait**. The buttons are disabled, the command sent to ECL^{i}PS^e side on the `debug_input` queue using **flush**. This is the asynchronous sending of data on the asynchronous queue: control is *not* handed over to ECL^{i}PS^e to process this command. Instead, the execution on the Tcl side carries on (and happens to finish immediately after the **flush**. Control is then returned to the ECL^{i}PS^e side as the Tcl handler has finished, and the ECL^{i}PS^e side continues execution after the `flush(debug_traceline)` goal. Next, `debug_input` is read for the tracer command, and this command is acted on.

Using the queue synchronously: to-ECL^{i}PS^e If the Tcl remote interface command **ec_stream_input_popup** (see section 6.6.2) is used to send data to the ECL^{i}PS^e-side (in section 6.6.2, the command was initiated by a read operation on the ECL^{i}PS^e side; here the command is invoked directly when Tcl side has control), then the following is a possible ECL^{i}PS^e handler:

```
Tcl code:

;# create the asynchronous queue, with
;#  from-ECLiPSe Tcl consumer handler: data_to_window
;#  to-ECLiPSe ECLiPSe handler event:  flush_myqueue
ec_async_queue_create myqueue {data_to_window textwin} flush_myqueue

...
;# get input for the queue and send to ECLiPSe side
ec_stream_input_popup "Data:" [ec_channel_to_streamnum myqueue]
...

ECLiPSe code:

:- set_event_handler(flush_myqueue, read_remote_data/2).

% Len is known when ec_stream_input_popup is used to send data
read_remote_data(_Event, rem_flushio(Queue,Len)) :-
read_string(Queue, "", Len, Data),
process(Data).
```

The ECL^{i}PS^e code defines `read_remote_data/2` as the handler for to-ECL^{i}PS^e data sent with **ec_flush** on the Tcl side. This handler is called when control is handed over to ECL^{i}PS^e side to read the data. Both the two optional arguments are used in this handler. The second argument supplies the ECL^{i}PS^e stream number for the queue and the length of data written. As the data is sent by explicitly calling **ec_stream_input_popup**, the length of the data sent is known, so

`read_string/4` can be used to read the exact amount of data. In the asynchronous queue, it is generally the programmer's responsibility to ensure that the read will not block.

Using the queue asynchronously: from-ECL^{i}PS^e The example `ec_async_queue_create` also defines a Tcl data consumer handler to handle data sent on the from-ECL^{i}PS^e direction, with a user defined argument of the text window that the data will be sent to. Here is a simple procedure which reads the data on the queue and places it on the text window specified:

```
Tcl code:

proc data_to_window {Window Stream} {
    set channel [ec_streamnum_to_channel $Stream]

    $Window insert end [read $channel]
}
```

The *Stream* argument is appended by the interface when the handler is invoked, and is the ECL^{i}PS^e stream number of the queue. The procedure simply reads the data from the corresponding Tcl channel and display the data on *Window*, the text window specified by the programmer.

6.6.4 Reusable Queue Names

ECL^{i}PS^e stream names are global in scope, so using fixed queues names like 'myqueue' might cause name conflicts with other modules, if the programmer intend the remote Tcl code to be usable with other ECL^{i}PS^e code. One way to avoid name clashes is to dynamically composing queue names using the name of the control connection:

```
Tcl code:
        append queue_name [ec_control_name] myqueue
        ec_queue_create $queue_name fromec {ec_stream_output_popup red textwin}
```

The user specified name 'myqueue' is appended to the control name of the remote connection to give a unique queue name. On the ECL^{i}PS^e side, the code will also need to use the dynamic name:

```
        :- local variable(remote_control).

        ...
        % code fragment to remember the control name
        remote_connect(Addr, Control, _),
        setval(remote_control, Control),
        ...

        ...
        % code fragment to use the queue
        getval(remote_control, Control),
```

```
concat_atom([Control, myqueue], QName),
...
write(QName, hello), flush(QName),
...
```

6.6.5 Translating the Queue Names

The remote queues connecting ECL^{i}PS^e and Tcl are given different names on the two sides. The remote Tcl interface keeps track of the ECL^{i}PS^e names for the queues on the Tcl side. On the ECL^{i}PS^e side, the queue has a stream number, as well as possibly several symbolic aliases. The interface only keeps track of one symbolic name – the one that is supplied in *ec_queue_connect* and *ec_async_queue_create*. If the ECL^{i}PS^e stream number was supplied in these commands, then the stream number is also considered the symbolic name for the queue as well. The Tcl interface provides several commands to convert the names from one form to another:

ec_streamname_to_channel *eclipse_name*
: Returns the Tcl channel name for the remote queue with the symbolic name *eclipse_name*.

ec_streamnum_to_channel *eclipse_stream_number*
: Returns the Tcl channel name for the remote queue with the ECL^{i}PS^e stream number *eclipse_stream_number*.

ec_channel_to_streamnum *channel*
: Returns the ECL^{i}PS^e stream number for the remote queue with the Tcl channel name *channel*.

ec_streamname_to_streamnum *eclipse_name*
: Returns the ECL^{i}PS^e stream number for the remote queue with the symbolic name *eclipse_name*.

ec_stream_nr *eclipse_name*
: This is an alias for **ec_streamname_to_streamnum** for compatibility with embedded interface.

6.7 Additional Control and Support

The remote interface provides additional support for controlling the interaction of the Tcl and ECL^{i}PS^e sides, such as explicit transfer of control between ECL^{i}PS^e and Tcl, and the disconnection of the Tcl and ECL^{i}PS^e sides. The interface also provides support for special user-defined commands to be executed during these events.

6.7.1 Initialisation During Attachment

In an application, after the Tcl side has been attached, typically some application specific initialisation needs to be performed, such as setting up various data queues between the two sides, and defining the actions to take when the two sides are disconnected. On both sides, these initialisations can be performed immediately after attachment. On the Tcl side, such actions can be specified in the optional *init_command* argument of *ec_remote_init*. On the ECL^{i}PS^e side,

such actions can be specified in the 'InitGoal' (last) argument of *remote_connect/3*. InitGoal can be a built-in, or a user-defined goal.

6.7.2 Disconnection and Control Transfer Support

Disconnection should normally be performed when the ECL^{i}PS^e application has finished using the GUI provided by the particular attached remote process. The disconnection may be initiated from either side. In addition to cleaning up and closing all the remote queues connecting the two sides, the disconnection would trigger the execution of user definable procedures on both sides (through an event on the ECL^{i}PS^e side, and a call-back on the Tcl side), which can be used to perform extra application specific cleanup and shutdown routines.
For the transfer of control from Tcl to ECL^{i}PS^e and vice versa, user-definable call-backs are made. This is to enable to define application specific restrictions on what the GUI is allowed to do when the ECL^{i}PS^e side has the control (for example, the GUI may have a button that sends an rpc goal to ECL^{i}PS^e when pressed. Such a button could be disabled by the call-back when control is transferred to ECL^{i}PS^e and reenabled when control is transferred back to Tcl).
Note that there are two types of transfer of control from ECL^{i}PS^e to Tcl: 1) when the control is implicitly yielded (e.g. initiating I/O from ECL^{i}PS^e with Tcl, or returning after an rpc call); 2) when the control is handed over by yielding explicitly (e.g. by calling **remote_yield/1** in ECL^{i}PS^e). With implicit yield, the Tcl side is expected to eventually handed back control implicitly to ECL^{i}PS^e, and not to explicitly hand control over to ECL^{i}PS^e before this. Thus two call-backs are provided when control is yield to Tcl: one is executed whenever the control is yielded, and the other is only executed when the control is explicitly yielded. Thus when control is explicitly yielded, both call-backs are executed. This can be useful for example by defining the explicit yield call-back to enable a button on the Tcl side that will explicitly transfer control back to ECL^{i}PS^e when pressed, which should only be enabled when ECL^{i}PS^e explicitly yielded to Tcl.
On the ECL^{i}PS^e side, an event is raised when the two sides disconnect. The event's name is the control stream's name. The user can define a handler for this event to allow user-defined action to take place on the ECL^{i}PS^e side on disconnection. The simplest way to define this handler is to do it during the connection, via the last argument of **remote_connect/3**.

Tcl side

ec_running
: checks if the ECL^{i}PS^e side has control. Returns 1 if ECL^{i}PS^e side has control, 0 otherwise. If that is the case, then the Tcl side cannot issue an ec_rpc goal. Note that ec_running will return 1 before connection and after disconnection.

ec_connected
: checks if the Tcl side is currently attached to ECL^{i}PS^e. Returns 1 if there is a connection to ECL^{i}PS^e (i.e. it is attached), 0 otherwise.

ec_resume
: explicitly hand-over control to ECL^{i}PS^e. Tcl side must have control when this command is called (i.e. *ec_running* must be false). This command returns when ECL^{i}PS^e side yields the control back to the Tcl side. Meanwhile, the Tcl process is not suspended as the Tcl event loop is entered while waiting for the yield.

ec_running_set_commands *?start? ?end? ?yield? ?disconnect?*

set up commands that will be called just before control is handed over to ECL^{i}PS^e (*start*), when control is handed back from ECL^{i}PS^e (*end*), when ECL^{i}PS^e explicitly yields control (*yield*), and when the Tcl side is disconnected by the ECL^{i}PS^e side (*disconnect*). The *start* and *end* commands are called both when control change hands explicitly (e.g. via *ec_resume*), or implicitly (e.g. by making an rpc call or performing I/O on a synchronous remote queue). An explicit yield from ECL^{i}PS^e will in addition call the *yield* command, *after* the *start* command is executed.

The default for each command is that no command will be called.

ec_disconnect *?side?*

disconnect the Tcl process from the ECL^{i}PS^e process. This closes all the connections between the two sides. The ECL^{i}PS^e side will abort from what it was doing. After disconnection, the two sides can no longer communicate, and *ec_running* will be set. The optional argument *side* specifies which side, tcl or eclipse, initiated the disconnection. For user's Tcl program, this will normally be the default tcl. If the disconnect is initiated from the Tcl side, this command will cause the ECL^{i}PS^e side to also close its connections to this remote connection, as well as raising the disconnect event in ECL^{i}PS^e associated with this remote connection. If the disconnect was initiated from the ECL^{i}PS^e side, then *ec_disconnect* will be called automatically with *side* set to eclipse, and the disconnect command set up by *ec_running_set_commands* will be executed.

ECL^{i}PS^e side

remote_yield(+Control)

Explicitly yields control from ECL^{i}PS^e to the remote side with the control stream *Control*. ECL^{i}PS^e execution will suspend until control is transferred back to ECL^{i}PS^e. This predicate returns when ECL^{i}PS^e side resumes control.

remote_disconnect(+Control)

Initiates disconnection from the remote side specified by *Control*. This will close all connections between ECL^{i}PS^e and the remote side, on both sides. It will also cause an event *Control* to be raised.

Note that if the ECL^{i}PS^e process is halted normally, then ECL^{i}PS^e will try to disconnect from every remote side it may be connected to.

6.8 Example

Figures 6.1 and 6.2 shows a simple example of the use of the interface. To try this program, the user should start an ECL^{i}PS^e, compile the ECL^{i}PS^e program. This will suspend on the `remote_connect/3`, waiting for the Tcl program to attach. The Tcl program should then be started on the same machine, and the attachment will be connected using the fixed host and port address. During the initialisation when attaching, a from-ECL^{i}PS^e queue is created, which is set to flush at every newline. This is done by an **ec_rpc** goal after the queue is created. As no Tcl data consumer handler is specified, the default Tcl data consumer handler **ec_stream_output_popup** handles and display the data on the Tcl side on a pop-up window.

```
% Tcl side will create the from-ECLiPSe queue gui_output, which will also
% automatically flush at the end of every line (and so transfer the data
% to the Tcl side)


disconnect_handler :-  % just terminate ECLiPSe execution
        writeln("Terminating...."),
        halt.


:-      remote_connect(localhost/5000, control,
        set_event_handler(control, disconnect_handler/0)),
        % for simplicity, the host and port are fixed.
        % Name for control connection is also fixed.
        % when remote_connect returns, the gui_output queue will be
        % connected to Tcl side already.
        % eclipse side initialisation follows is just to set up the
        % handler for disconnection....
        writeln(gui_output, "Connected...").

% this is for yielding control to Tcl. No need to specify Control explicitly
tclyield :-
        remote_yield(control).
```

Figure 6.1: Example use of interface: ECL^{i}PS^e code

The Tcl side GUI has just one button which allows the application to terminate. This button is disabled when ECL^{i}PS^e side has control, and this is done by setting up the appropriate call-backs to disable and enable the button in *ec_running_set_commands* on the Tcl side.
On disconnection, which can be initiated either by pressing the button on the Tcl side, or by quitting from ECL^{i}PS^e, the handlers for disconnection ensures that both the ECL^{i}PS^e and Tcl program terminates.

6.9 Differences From the Tcl Embedding Interface

The remote Tcl interface is designed to be largely compatible with the embedded Tcl interface, so that a user GUI can be written that allows either interfaces to be used, while sharing most of the code in both ECL^{i}PS^e and Tcl. An example of this is the Tkeclipse development tools. Some remote specific code would need to be written. This includes code to handle the connection and disconnection of the Tcl and ECL^{i}PS^e sides: there is no equivalent to ec_cleanup in the embedded Tcl interface, as the termination of the ECL^{i}PS^e side should be handled in ECL^{i}PS^e. In addition, the user may need to provide code to restrict the interaction within the Tcl/Tk GUI when control is transferred to ECL^{i}PS^e. Aside from this, the rest of the code should be reusable, if the user exercises some care.
The supported compatible methods of communicating between Tcl and ECL^{i}PS^e in the two interfaces is via the **ec_rpc** calls and the use of the I/O queues. The **ec_rpc** mechanism should

behave the same in both interfaces. For the queues, there are some differences because the queues are in-memory queues in the embedded interface, but are socket channels in the remote interface. This leads to the following differences:

- Data will not appear on the other side until the queue is flushed on the side that is generating the output. To make code compatible, output queues should be always flushed.
- In general, reading data via a blocking socket requires the size of the data to be explicitly specified, except when I/O is done via the EXDR primitives, where size is implicitly specified. Only EXDR format is supported when data is sent from Tcl to ECL^{i}PS^e. That is, a write handler for a Tcl write channel must write the data in EXDR format. For output from the ECL^{i}PS^e side, the Tcl read handler would be supplied with the length of the output when it is invoked, and this information must be used if the Tcl read is not done via an EXDR primitive. However, in general it is strongly suggested that only EXDR formatted data should be sent via the queues in both direction.
- An I/O operation on the stream may block if there is no handler to consume/produce the data on the other side. If a handler is specified via the *command* argument, then a corresponding handler in the Tcl side will be invoked at the correct place when the ECL^{i}PS^e side produces output or request input.
- The channel identifier is not of the form `ec_queueX`. To make code portable, the name of the channel should be obtained from the ECL^{i}PS^e stream symbolic name or number via the commands **ec_streamname_to_channel** or **ec_streamnum_to_channel**.
- Asynchronous queues are bi-directional. In the embedding interface, there are no asynchronous queues, and **ec_async_queue_create** is aliased to **ec_queue_create**, and a unidirectional queue is created. Thus for compatibility, these queues should only be used in one direction.

In summary, to write code that will work for both the remote and embedded interfaces, the data should be sent using EXDR format, flush always performed, and a handler (*command* argument) provided. The Tcl channel identifier should not be constructed explicitly.
However, there may be cases where the two interfaces need to be distinguished. For example, if the Tcl side is to perform some operations on the file system for the ECL^{i}PS^e side (e.g. selecting a file via a GUI), then with the remote interface, the two sides might not have access to the same file systems, and being able to distinguish whether the interface is remote or embedded allows the user to provide code to handle this.
To obtain information on which interface is being used, use the command:

ec_interface_type
: returns remote for the remote interface, and embedded for the embedding interface.

```
# ECLIPSEDIR has to be set to where your ECLiPSe is at
lappend auto_path [file join $ECLIPSEDIR) lib_tcl]

package require remote_eclipse

proc terminate {} {
    destroy .
}

# disable the terminate button when control is transferred to ECLiPSe
proc disable_button {} {
    .b configure -state disabled
}

# enable the terminate button when control is transferred back to Tcl
proc enable_button {} {
    .b configure -state normal
}

# the initialisation procedure, called when the remote side is attached.
# this creates the gui_output from-ECLiPSe queue, and then uses an
# ec_rpc to cause the queue to flush at every newline
proc initialise {} {
ec_queue_create gui_output fromec
ec_rpc "set_stream_property(gui_output, flush, end_of_line)"
}

# this button initiates disconnection and terminates the Tcl program
pack [button .b -text "Terminate" -command "ec_disconnect; terminate"]
# only start and end commands given; the others default to no commands
ec_running_set_commands disable_button enable_button

# disable button initially as ECLiPSe side has initial control
disable_button

# the attachment to ECLiPSe includes the initialisation ec_init, which
# creates the gui_output queue. The connection is at the fixed port address
ec_remote_init localhost 5000 initialise
# Tcl side initially has control, hand it over to ECLiPSe...
ec_resume resume
```

Figure 6.2: Example use of interface: Tcl code

Chapter 7

Tcl Peer Multitasking Interface

7.1 Introduction

This chapter describes how to use the peer multitasking interface with the Tcl peer interfaces. The usage is the same regardless of if the Tcl interface is of the embedded or remote variants. The facilities described here allows a Tcl peer to participate in peer multitasking, so that different peers can apparently interact with ECL^{i}PS^e simultaneously. This would for example allow GUI windows that are implemented using different peers for the same ECL^{i}PS^e process to be appear as if they can interact with the host eclipse side at the same time.

7.2 Loading the interface

The peer multitasking interface is provided as a Tcl-package called **eclipse_peer_multitask**, and must be loaded along with either the embedded (**eclipse**) or remote (**remote_eclipse**) variant of the Tcl peer interface.

```
lappend auto_path "/location/of/my/eclipse/lib_tcl"
package require remote_eclipse
package require eclipse_peer_multitask
```

7.3 Overview

A peer must already exist before it can participate in peer multitasking: (i.e. it has already been set up using ec_init (embedded) or ec_remote_init (remote)).
Peer multitasking occur in a *multitasking phase*, which is a special way for an ECL^{i}PS^e side to hand over control to the external side. Effectively, instead of handing over control to a single peer, control is handed over repeatedly to all the peers that are participating in peer multitasking. The control is shared between these peers in a round-robin fashion, giving rise to a form of co-operative multitasking. The multitasking is co-operative in that each peer has to give up control, so that the control can be passed to the next multitasking peer. A peer multitasking phase is started by calling the predicate **peer_do_multitask/1** in ECL^{i}PS^e.
To participate in peer multitasking, a peer must be "registered" (initialised) for peer multitasking. This is done by executing the procedure **ec_multi:peer_register** from Tcl. Once registered, the peer will take part in subsequent multitasking phases.

The registration can set up three user-defined handlers: the **start** handler, the **interact** handler, and the **end** handler. During a multitasking phase, control is handed over to a multitasking peer, which invokes one of these handlers, and when the handler returns, control is handed to the next multitasking peer. The interact handler is normally invoked, except at the start (first) and end (last) of a multitasking phase. In these cases, the start and end handlers are invoked respectively.
A 'type' (specified in the call to **peer_do_multitask/1**) is associated with every multitasking phase to allow multitasking phases for different purposes (distinguished by different 'type's). This type is passed to the handlers as an argument. At the start of a multitasking phase, a peer should indicate if it is interested in this particular multitasking phase or not. If no peer is interested in the multitasking phase, then the multitasking phase enters the end phase after the initial round of passing control to all the multitasking peers, i.e. control is passed one more time to the multitasking peers, this time invoking the end handler. If at least one peer indicates that it is interested in the multitasking phase, then after the initial start phase, the control will be repeatedly handed over to each multitasking peer by invoking the interact handler. This phase is ended when one (or more) peer indicates that the multitasking phase should end (at which point the end phase will be entered in which the end handler will be invoked for each multitasking peer).

7.3.1 Summary of Tcl Commands

Here is a more detailed description of the Tcl procedures:

ec_multi:peer_register *?Multitask handlers*

Registers for peer multitasking, and setup the (optional) multitask handlers. There handlers can be specified: a) start, invoked at the start of a multitasking phase; b) end, invoked at the end of a multitasking phase, and c) interact, invoked at other times when the peer is given control during a multitasking phase. **Multitask handlers** is a list specifying the handlers, where each handler is specified as two list elements of the form: `type name`, where type is either **start**, **end** or **interact**, and name is the name of the user defined handler. For example:

```
ec_multi:peer_register [list interact tkecl:multi_interact_handler
      start tkecl:multi_start_handler end tkecl:multi_end_handler]
```

When control is handed over to the peer during a peer multitasking phase, the appropriate handler (if defined) is invoked. When the handler returns, control is handed back to ECL^{i}PS^e (and passed to the next multitasking peer). Note that events are not processed while the peer does not have control. The Tcl command **update** is therefore called each time the peer is given control, to allow any accumulated events to be processed. As long as the peer is given control frequently enough, then any interactions with the peer would appear to be continuous.

The handlers are invoked with the 'type' of the multitasking phase supplied as the first argument. This will for example allow the start handler to determine if it is interested in this multitasking phase. They can also set one of the following return codes:

continue indicates that the peer wants to continue the multitasking phase. In particular, this should be returned by the start handler if it is interested in the multitasking

phase. Note that the multitasking phase is not guaranteed to continue, as it can be terminated by another multitasking peer.

terminate indicates the termination of a multitasking phase. The multitasking phase will enter the end phase, where the end handlers will be invoked once per peer before the multitasking phase finishes.

For example, here is the start handler used in the Tk development tools:

```
proc tkecl:multi_start_handler {type} {

    switch $type {
        tracer {
            # multitasking phase is 'tracer'
            # only do handling of port if the tracer window exists
            if [winfo exists .ec_tools.ec_tracer] {
                tkecl:handle_tracer_port_start
                set of_interest  continue
            }
        }
        default {
            set of_interest no
            # do nothing
        }
    }

    return $of_interest
}
```

An error is raised if this procedure is called while the peer is already registered for multitasking.

ec_multi:peer_deregister

Deregisters peer from multitasking. After calling this procedure, the peer will not participate in any further multitasking. If this procedure is called during a multitasking phase, the peer will not participate further in that multitasking phase once control is returned. The multitasking phase will continue, unless there are no multitasking peers left, in which case an error will be raised on the ECL^{i}PS^e side. The peer multitask control queue will be automatically closed.

An error is raised if this procedure is called while the peer is not registered for multitasking.

ec_multi:get_multi_status

Returns the current peer multitasking status for the peer. The values can be:

- not_registered: the peer is not registered for peer multitasking.
- off: the peer is registered for peer multitasking, but is not currently in a multitasking phase.

- on: the peer is registered for peer multitasking, and is currently in a multitasking phase.

Note that the peer multitasking code is implemented using peer queue handlers. Thus, the peer multitask status is set to 'on' before the multitask start handler is called, but *after* the 'ECL^{i}PS^e end' handler. Conversely, the peer multitask status is set to 'off' after the multitask end handler, but *before* the ECL^{i}PS^e start handler.

7.4 Example

Here we present a simple example use of the Tcl peer multitasking facilities. The full programs (Tcl and ECL^{i}PS^e code) are available in the ECL^{i}PS^e distribution as `example_multi.ecl` and `example_multi.tcl`.
The Tcl program uses the remote Tcl peer interface to create a window that interacts with the ECL^{i}PS^e process it is attached to. Multiple copies of the program can be run, and attached to the same ECL^{i}PS^e process with a different remote peer. Each window has three buttons:

- run: a button to send an ERPC goal to ECL^{i}PS^e (in this case a simple writeln with the peer name;
- end: a button to end interaction with ECL^{i}PS^e and return control to ECL^{i}PS^e;
- reg: a button to toggle the registration/deregistration of the peer for peer multitasking;

The program interacts with ECL^{i}PS^e when the run button is pressed. This can be done either during a peer multitasking phase, or when the program's specific peer is given control on its own. When the peer is given control on its own, only it can interact with ECL^{i}PS^e; while during peer multitasking, all the multitasking peers (the copies of the program that are running) can interact with ECL^{i}PS^e, i.e. the run buttons in all the windows can all be pressed.
After attaching to ECL^{i}PS^e with **ec_remote_init**, the program sets various handlers for the handing of control between ECL^{i}PS^e and itself with **ec_running_set_commands**:

```
ec_running_set_commands ec_start ec_end {} disable_buttons
```

The handlers for when control is handed back to ECL^{i}PS^e, `ec_start`, and when control is handed over from ECL^{i}PS^e, `ec_end`, are defined thus:

```
proc ec_end {} {
    if {[ec_multi:get_multi_status] != "on"} {
        enable_buttons
    }
}

proc ec_start {} {
    if {[ec_multi:get_multi_status] != "on"} {
        disable_buttons
```

```
    }
}
```

`enable_buttons` and `disable_buttons` enables and disables the buttons for the window, respectively. The code tests if the peer is multitasking with `ec_multi:get_multi_status`. This is needed because during a peer multitasking phase, control is repeatedly handed back and forth, and we don't want the buttons to be repeatedly enabled and disabled during this phase.
Next, the program registers the peer for peer multitasking:

```
    ec_multi:peer_register [list start multi_start_handler \
                            interact multi_interact_handler]
```

No special handler is needed for the end of multitasking, as no special action is needed beyond disabling the buttons. The return code is stored in a global variable `return_code`. The start handler is defined thus:

```
proc multi_start_handler {type} {
    global return_code

    if {$type == "demo"} {
        set return_code continue
        enable_buttons
    } else {
        set return_code no
    }
    return $return_code
}
```

As discussed, multitasking phases can be of different types. For demonstrating this multitasking features of this example program, the type is "demo". Therefore the handler tests for this and enables the button if the phase is "demo". On the ECL^{i}PS^e side, the multitasking phase is started with the following predicate call:

```
        peer_do_multitask(demo),
```

The interact handler is defined thus:

```
proc multi_interact_handler {type} {
    global return_code

    if {$return_code == "terminate"} {
            disable_buttons
    }
    return $return_code
```

The code checks for the two cases where the user has requested to terminate the multitasking phase by pressing the `.end` button, and disables the buttons. The end button itself invokes the following code to set `return_code`:

```
proc end_interaction {} {
    global return_code
    set return_code terminate
    if {[ec_multi:get_multi_status] != "on"} {
        ec_resume
    }
}
```

The code checks if it is in a peer multitasking phase, and if so, `return_code` is set to `terminate`, so that when the handler returns, the multitasking phase will terminate. Otherwise, the peer has been explicitly handed control exclusively, and so control is handed back to ECL^{i}PS^e in the normal way using **ec_resume**.
The program also allows a peer to deregister from multitasking or, if already deregistered, to register again for multitasking. This is handling by the following two procedures:

```
proc register_for_multi {} {
    ec_multi:peer_register [list start multi_start_handler]
    .reg configure  -command {deregister_for_multi}
    .reg configure -text "Deregister multitasking"
}

proc deregister_for_multi {} {
    ec_multi:peer_deregister
    .reg configure  -command {register_for_multi}
    .reg configure -text "Register multitasking"
}
```

Pressing the `.reg` button will either call `register_for_multi` or `deregister_for_multi`, depending on if the peer is currently deregistered or registered for peer multitasking (respectively). The procedure also changes the button to toggle to the other state.
Pressing the button during a peer multitasking phase will remove the peer from multitasking immediately. If pressed while the peer is given exclusive control, the peer will not participate in future multitasking phase (unless it is re-registered).

Chapter 8

Using the Java-ECL^{i}PS^e Interface

The Java-ECL^{i}PS^e Interface is a general-purpose tool for interfacing ECL^{i}PS^e with Java, Sun's popular object-oriented platform-independent programming language. It could be used for example to write a Java graphical front-end to an ECL^{i}PS^e optimisation program or to interface ECL^{i}PS^e with a commercial application written in Java. This document assumes a moderate level of programming in Java (e.g. how to use classes, packages, interfaces etc. and some knowledge of the *java.util*, *java.io* and *java.net* libraries) and a basic knowledge of how to interact with ECL^{i}PS^e (e.g. how to execute a goal, nondeterminism, what lists are etc.). There are also some diagrams in UML notation, but these are not crucial to the reader's understanding.
Section 8.1 is intended to get you started quickly by running the example program `QuickTest.java`. Section 8.2 gives a brief overview of the functionality of the Java-ECL^{i}PS^e interface by taking a closer look at how `QuickTest.java` works. In the other sections of the document you will learn how to write Java programs which can:

- Manipulate ECL^{i}PS^e terms and other data in Java (Section 8.3).
- Execute goals in ECL^{i}PS^e and use the results of this computation in Java (Section 8.4).
- Transfer data between Java and ECL^{i}PS^e using queues. (Section 8.5).
- Initialise and terminate different kinds of connections between ECL^{i}PS^e and Java. (Section 8.7).

Throughout this document, `<eclipse_dir>` stands for the root location of your ECL^{i}PS^e installation. Detailed `javadoc`-generated HTML descriptions of the API provided by the `com.parctechnologies.eclipse` package can be found at:

```
<eclipse_dir>/doc/javadoc/JavaEclipseInterface/index.html
```

You may wish to browse the API documentation after reading this document to obtain more detailed descriptions of classes and interfaces in the package.

8.1 Getting Started

At the end of this section you will run the simple Java program `QuickTest.java` which uses ECL^{i}PS^e. First of all though you need to check that your Java SDK version is recent enough and that your classpath correctly set up.

8.1.1 Check your Java SDK version

Use of the Java-ECL^{i}PS^e Interface requires an installation of the Java SDK (Standard Developer's Kit) version 1.2.2 or later. If your Java SDK installation is an earlier version than this or you do not have the Java SDK on your machine, the latest version can be downloaded from Sun Microsystems Inc. (`http://www.sun.com`).

8.1.2 Make the `com.parctechnologies.eclipse` package available in your class path

The Java-ECL^{i}PS^e Interface consists mainly of a Java package which is used as a library by the Java programs you will write. This package is included as a `.jar` file located within the ECL^{i}PS^e distribution at:

```
<eclipse_dir>/lib/eclipse.jar
```

You are free to copy `eclipse.jar` to a more convenient location. However, to compile or run any Java programs which use the package you must include the full path of `eclipse.jar` in your classpath. For more information on using the classpath, please consult your Java documentation.

8.1.3 Compile and run `QuickTest.java`

To test that everything is working as it should be, and to see a quick example of the Java-ECL^{i}PS^e Interface at work, try compiling and running the Java program `QuickTest.java`. This starts up an ECL^{i}PS^e from Java and tells it to write a message to `stdout`. The program can be found at

```
<eclipse_dir>/doc/examples/JavaInterface/QuickTest.java
```

After compilation, to run the program, start the Java interpreter as you normally would but before the name of the class, supply the command line option

```
-Declipse.directory=<eclipse_dir>
```

This tells Java where to find the ECL^{i}PS^e installation, so it can run ECL^{i}PS^e. **You should use this command line options when running all other examples in this document**. When you run `QuickTest.java`, you should get a single line of output: `hello world`. How `QuickTest.java` works is explained in Section 8.2.

8.2 Functionality overview: A closer look at `QuickTest.java`

To give a broad overview of how the Java-ECL^{i}PS^e Interface works, we take a closer look at `QuickTest.java`. Here is the Java source code from `QuickTest.java`.

```
import com.parctechnologies.eclipse.*;
import java.io.*;

public class QuickTest
{
  public static void main(String[] args) throws Exception
```

```
  {
    // Create some default Eclipse options
    EclipseEngineOptions eclipseEngineOptions = new EclipseEngineOptions();

    // Object representing the Eclipse engine
    EclipseEngine eclipse;

    // Connect the Eclipse's standard streams to the JVM's
    eclipseEngineOptions.setUseQueues(false);

    // Initialise Eclipse
    eclipse = EmbeddedEclipse.getInstance(eclipseEngineOptions);

    // Write a message
    eclipse.rpc("write(output, 'hello world'), flush(output)");

    // Destroy the Eclipse engine
    ((EmbeddedEclipse) eclipse).destroy();
  }
}
```

The structure of the `main` method in `QuickTest.java` contains elements which will appear in any Java code which uses the Java-ECL^{i}PS^e interface. These are as follows:

Always import the `com.parctechnologies.eclipse` package

Note the first line using `import`. We need to have this in every Java source file which uses the Java-ECL^{i}PS^e Interface so that it can load classes from the package.

Declare a reference to an *EclipseConnection* or an *EclipseEngine*

EclipseConnection and its subinterface *EclipseEngine* are Java interfaces which contain the methods used when interacting with ECL^{i}PS^e.

Create an object which implements *EclipseConnection* or *EclipseEngine*

There are a number of classes which implement these interfaces. In the case of `QuickTest.java` we use an instance of *EmbeddedEclipse* . The initialisation of these implementing classes varies from class to class. In the case of *EmbeddedEclipse* initialisation is done by creating and configuring an *EclipseEngineOptions* object and using this in an invocation of the *EmbeddedEclipse* class' static method `getInstance`.

Interact with ECL^{i}PS^e using methods in the *EclipseConnection* or *EclipseEngine* interface

We interact with the ECL^{i}PS^e engine by invoking methods on the object which implements *EclipseConnection* or *EclipseEngine* . In the case of `QuickTest.java` we invoke the `rpc` method, which causes the ECL^{i}PS^e to execute a goal, in this case one which simply prints a message.

Terminate interaction with the ECL^{i}PS^e

In order to clean up, after the Java program has finished interacting with ECL^{i}PS^e, the interaction should be terminated. Like initialisation, how this termination is done varies among the classes which implement the *EclipseConnection* and *EclipseEngine* interfaces. In the case of `QuickTest.java`, termination is done by invoking the `destroy` method on the *EmbeddedEclipse* object.

8.3 Java representation of ECL^{i}PS^e data

The Java-ECL^{i}PS^e Interface uses a set of conventions and Java classes so that data types common in ECL^{i}PS^e can be represented. Representing ECL^{i}PS^e data types is useful for:

- Constructing compound goals to be sent to ECL^{i}PS^e for execution.
- Deconstructing the results of ECL^{i}PS^e's computation, which are returned as a compound goal.
- Communicating compound terms and other data via queues.

More details on these tasks will be provided in Sections 8.4 and 8.5, but it is first necessary to understand how ECL^{i}PS^e data is represented in Java.

8.3.1 General correspondence between ECL^{i}PS^e and Java data types

Not all ECL^{i}PS^e data types are represented: for example, at present the ECL^{i}PS^e type `rational` has no Java equivalent. However, all the most useful ECL^{i}PS^e types have a corresponding Java class. Table 8.1 gives the general correspondence between those ECL^{i}PS^e data types which are supported and the Java classes or interfaces which are used to represent them. The ECL^{i}PS^e types/Java classes which appear in the table are those which map to or from the ECL^{i}PS^e external data representation (EXDR) definition.

ECL^{i}PS^e data type	Java class/interface
`atom`	*Atom*
`compound`	*CompoundTerm*
`integer`	*java.lang.Integer* *java.lang.Long*
`list`	*java.util.Collection*
`float`	*java.lang.Double* *java.lang.Float*
`string`	*java.lang.String*
`variable`	*null*

Table 8.1: The correspondence between ECL^{i}PS^e and Java data types. The Java classes are within the `com.parctechnologies.eclipse` package unless otherwise specified.

The general rule is that you can send data to ECL^{i}PS^e by passing the relevant method an instance of the Java class corresponding to the ECL^{i}PS^e data type you want on the ECL^{i}PS^e

side. When data comes back from ECL^{i}PS^e it will be an instance of *java.lang.Object* but this can be cast to the relevant Java class, which must either be known beforehand or determined e.g. using the `getClass()` method.
There are also a number of peculiarities for certain cases which we now explain.

8.3.2 Atoms and compound terms

Atoms are simple: these are constructed in Java using the constructor of the *Atom* class: the string parameter of the constructor becomes the atom symbol. Although the Java interface *CompoundTerm* is listed above, compound terms (except lists) are usually constructed using the *CompoundTermImpl* class, which implements *CompoundTerm*. Incidentally, *Atom* also implements *CompoundTerm*, even though strictly speaking ECL^{i}PS^e atoms are not compound. Here is an example of *CompoundTermImpl* at work:

```
// Construct a term in Java to represent the Eclipse term foo(a, b, 3).
private static CompoundTerm construct_term()
{
  Atom a = new Atom("a");
  Atom b = new Atom("b");
  Integer numberThree = new Integer(3);
  CompoundTerm theTerm = new CompoundTermImpl("foo", a, b, numberThree);

  return(theTerm);
}
```

This method is taken from the example Java program `DataExample1.java` which can be found in the examples directory `<eclipse_dir>/doc/examples/JavaInterface`. The rest of the program sends ECL^{i}PS^e a goal which tells it to write the term created by `construct_term()` to `stdout`.
In this example we use an *CompoundTermImpl* constructor whose first parameter is a string which becomes the functor of the term. The remaining parameters are instances of *java.lang.Object*. They may be instances of any class or interface which appears in Table 8.1. These become the arguments of the term. *CompoundTermImpl* has a number of other convenient constructors. See the API documentation for details of these.
Note that the object returned by `construct_term()` is declared not as a *CompoundTermImpl* but as a *CompoundTerm*. CompoundTerm is the Java interface for objects representing compound terms. Anything which implements CompoundTerm can be sent to ECL^{i}PS^e as a compound term.
Instead of using *CompoundTermImpl*, you may wish to implement *CompoundTerm* yourself. The benefit of this is that you can pass any object implementing *CompoundTerm* to an `rpc` invocation, and it can supply a functor and arguments without the unnecessary creation of another object. To do this you may wish to subclass *AbstractCompoundTerm*.

8.3.3 Lists

Whenever you want to construct a list for ECL^{i}PS^e or deconstruct a list coming from ECL^{i}PS^e, you use the *java.util.Collection* interface. Look at the following method, taken from the example Java program `DataExample2.java` (which can be found in the examples directory `<eclipse_dir>/doc/examples/JavaInterface`).

```
// Construct a collection in Java to represent the Eclipse
// list [1, foo(3.5), bar].
private static Collection construct_collection()
{
    Collection theCollection = new LinkedList();

    theCollection.add(new Integer(1));
    theCollection.add(new CompoundTermImpl("foo", new Double(3.5)));
    theCollection.add(new Atom("bar"));

    return(theCollection);
}
```

If you study, compile and run `DataExample2.java` you will see that the collection is indeed translated into the required ECL^{i}PS^e list. You will also see that order is maintained in the sense that the order of elements as they appear in the ECL^{i}PS^e list will equal the collection's iterator order (the converse is true if the data is coming from ECL^{i}PS^e to Java).
Also note that the ECL^{i}PS^e empty list (`[]`) is represented in Java by the constant *java.util.Collections.EMPTY_LIST*.

8.3.4 Floats and Doubles

The ECL^{i}PS^e data type `float` is always converted to *java.lang.Double* in Java. However, ECL^{i}PS^e can be sent an instance of *java.lang.Double* or *java.lang.Float*: both will be converted to `float` in ECL^{i}PS^e. One value of *java.lang.Double* and *java.lang.Float* has no counterpart in ECL^{i}PS^e: the not-a-number (NaN) value. Infinite values can be sent in either direction.

8.3.5 Integers

ECL^{i}PS^e can be sent instances of either *java.lang.Integer* (32-bit integers) or *java.lang.Long* (64-bit integers). Both of these will be translated to type `integer` on the ECL^{i}PS^e side. When ECL^{i}PS^e sends data to Java, it will decide between the two classes depending on the number of bits needed for the integer. So for example, if the number is small enough to fit in an Integer, that is what will be returned. Note that therefore, the type of number coming back from ECL^{i}PS^e cannot be relied upon to be of one type or the other if it could fall on either side of the 32-/64-bit boundary.
If you require a set of numbers coming from ECL^{i}PS^e to be all of one Java type e.g. long, then the best approach is to cast the object returned by ECL^{i}PS^e to an instance of *java.lang.Number* and then invoke the appropriate conversion method e.g. `longValue()`.

8.3.6 Variables

The Java *null* token is used to represent any variables being sent to ECL^{i}PS^e. All variables coming from ECL^{i}PS^e will appear as *null*. The limitations of this are discussed in more detail in Section 8.4.

8.3.7 The `equals()` method

The `equals()` method has been overridden for *AbstractCompoundTerm* and therefore also for *Atom* and *CompoundTermImpl*. The implementation returns `true` iff the parameter *Object* implements *CompoundTerm* and its functor and arity are equal to those of the *AbstractCompoundTerm*, and pairwise invocations of `equals()` return `true` between each of the *AbstractCompoundTerm*'s arguments and the corresponding argument of the parameter object.

8.4 Executing an ECL^{i}PS^e goal from Java and processing the result

The *EclipseConnection* interface provides a single method `rpc` (Remote Predicate Call) for executing goals in the ECL^{i}PS^e. This method takes as a parameter the goal to be executed. How to construct this parameter is dealt with in Section 8.4.1. Section 8.4.2 explains how to deal with the results returned by `rpc`. Finally, some more details about the execution of the goal are given in Section 8.4.3.

8.4.1 Passing the goal parameter to `rpc`

There are main two variants of `rpc` which differ in the class of the goal parameter.
The simplest way to use `rpc` is to pass it an instance of *java.lang.String* which should be the goal as it would be typed into the ECL^{i}PS^e command line. Just like with the ECL^{i}PS^e command line, the goal can be a conjunction of several subgoals. An example of this is the use of `rpc` in the example program `QuickTest.java`. Also please note a full stop is not necessary.
The string-parameter `rpc` variant is somewhat inefficient and it is also inflexible because creating goals dynamically, or goals involving strings is tricky. It should really only be used for short simple goals. A more flexible and efficient alternative is to invoke `rpc`, passing it a parameter object which implements the *CompoundTerm* interface (discussed in Section 8.3.2). In this case the term becomes the goal. Here is an example of using this version of `rpc`, taken from `DataExample1.java`.

```
CompoundTerm a_term = construct_term();

// Get Eclipse to write the term to stdout and flush
eclipse.rpc(
            new CompoundTermImpl(",",
                        new CompoundTermImpl("write",
                                    new Atom("output"), a_term),
                        new CompoundTermImpl("flush", new Atom("output"))
                        )
            );
```

Using this variant is a bit more cumbersome (e.g. the creation of a new *CompoundTermImpl* for the conjunction of goals in the above example) but it would be useful for large goals constructed dynamically. There are also a number of "convenience" `rpc` methods, where instead of providing a *CompoundTerm*, you provide the objects from which the term is made. See the API documentation for more details of these variants.

Note: yield/2, remote_yield/1 and rpc

The builtins **yield/2** and **remote_yield/1** should not be executed anywhere within an `rpc` goal, as they will cause the goal to return prematurely.

8.4.2 Retrieving the results of an rpc goal execution

The `rpc` method returns an object which implements *CompoundTerm*. This object is the Java representation of the goal term, with the solution substitution applied to its variables.
The solution substitution can be deconstructed using the returned *CompoundTerm*'s `arg` method. This method takes an integer (the argument position) and returns an Object which is the Java representation of the ECL^{i}PS^e argument at that position.
In the returned *CompoundTerm* instantiated variables are replaced with their instantiations. Hence even if the variable was named in the initial goal, its instantiation is identified in the returned goal by its *position* rather than its name. Uninstantiated variables in the returned *CompoundTerm* are represented using the Java *null* token.
If a variable in the `rpc` goal becomes instantiated to an ECL^{i}PS^e data type which does not have an equivalent EXDR type (such as breal), then in the returned *CompoundTerm* it will appear as the Java *null* token.
The following Java code is an example of how the returned *CompoundTerm* can be deconstructed to extract the variable substitutions.

```
...
    CompoundTerm result = eclipse.rpc("X = Q, Y is 2.1 + 7");

    // The top-level functor of the goal term is ",".
    // The first and second arguments of the goal term are the two subgoals
    // and we can safely cast these as CompoundTerms.
    CompoundTerm firstGoal = (CompoundTerm) result.arg(1);
    CompoundTerm secondGoal = (CompoundTerm) result.arg(2);
    // X is the first argument of the first goal.
    Object firstGoalFirstArg = firstGoal.arg(1);
    // Y is the first argument of the second goal.
    Object secondGoalFirstArg = secondGoal.arg(1);

    System.out.println("X = "+firstGoalFirstArg);
    System.out.println("Y = "+secondGoalFirstArg);
...
```

The output will be:

```
X = null
Y = 9.1
```

Other ways an rpc invocation can terminate

Apart from succeeding and returning a *CompoundTerm*, `rpc` can throw exceptions. If the goal fails, an instance of *Fail* is thrown. So to test for failure you must catch this exception. An instance of *Throw* is thrown if ECL^{i}PS^e itself throws an error.

8.4.3 More details about `rpc` goal execution

Variables

As explained in Section 8.3.6, ECL^{i}PS^e variables are always represented by the *null* token, when `rpc` is invoked with a *CompoundTerm* parameter. When used in an `rpc` goal, each *null* token represents a different variable. Using *CompoundTerm* you cannot represent a goal with multiple occurrences of a single variable. So, for example the following Java code will output `q(b, b)` for the first `rpc` invocation and `q(a, b)` for the second.

```
...

  eclipse.rpc("assert(q(a, b))");
  eclipse.rpc("assert(q(b, b))");

  System.out.println(eclipse.rpc("q(X, X)"));
  System.out.println(eclipse.rpc(new CompoundTermImpl("q", null, null)));

...
```

Nondeterminism

The `rpc` feature does not support the handling of nondeterminism in the execution of the ECL^{i}PS^e goal. If the goal succeeds, control is returned to Java immediately after the first solution to the goal is found in ECL^{i}PS^e. All choice-points thereafter are ignored. So, for example, although the first ECL^{i}PS^e goal below would leave choice-points, it would be equal in effect to the second.

```
...
result = eclipse.rpc("member(X, [1, 2, 3, 4, 5])");
...
result = eclipse.rpc("member(X, [1])");
```

This is not a practical problem. It merely implies that if you are using nondeterminism to generate multiple solutions, you should collect these on the ECL^{i}PS^e side using a meta-level built-in predicate such as `findall/3` and then return the result to Java.

Concurrent invocations of `rpc`

Note that the `rpc` method is `synchronized`. Therefore if, while one Java thread is currently executing an `rpc` invocation, a second Java thread tries to invoke the `rpc` method of the same *EclipseConnection*, the second thread will block until the first thread's `rpc` invocation returns.

Nested invocations of `rpc`

During the execution of the `rpc` method, control is transferred to ECL^{i}PS^e. Due to the *QueueListener* feature which is discussed in Section 8.5, control is sometimes temporarily returned to Java before the ECL^{i}PS^e execution has finished. It is possible for this Java code itself to invoke `rpc`, thus leading to nested `rpc` invocations. Nested `rpc` invocations should not cause any problems.

8.5 Communicating between Java and ECL^{i}PS^e using queues

In the Java-ECL^{i}PS^e Interface, *queues* are one-way data streams used for communication between ECL^{i}PS^e and Java. These are represented on the ECL^{i}PS^e side using "peer queues", which are I/O streams. The Java-ECL^{i}PS^e Interface includes the classes *FromEclipseQueue* and *ToEclipseQueue* which represent these queues on the Java side. *FromEclipseQueue* represents a queue which can be written to in ECL^{i}PS^e and read from in Java. A *ToEclipseQueue* is a queue which can be written to in Java and read from in ECL^{i}PS^e.

Section 8.5.1 discusses how queues are opened, referenced and closed from either the Java or ECL^{i}PS^e sides. We also discuss here how to transfer byte data in both directions. However, the programmer need not be concerned with low-level data operations on queues: whole terms can be written and read using the *EXDRInputStream* and *EXDROutputStream* classes discussed in Section 8.5.2.

Via the *QueueListener* feature, Java code can be invoked (in a sense) from within ECL^{i}PS^e. The use of this feature is discussed in Section 8.5.3. In some cases, the standard streams (`stdin`, `stdout` and `stderr`) of the ECL^{i}PS^e engine will be visible to Java as queues. How to use these is discussed in Section 8.5.4.

8.5.1 Opening, using and closing queues

We now explain the standard sequence of events for using queues. Opening and closing, can be performed in a single step from either the Java or the ECL^{i}PS^e side.

Opening a queue using Java methods

FromEclipseQueue and *ToEclipseQueue* do not have public constructors. Instead, we invoke `getFromEclipseQueue` or `getToEclipseQueue`. This asks the *EclipseConnection* object for a reference to a *FromEclipseQueue* or *ToEclipseQueue* instance which represents a new queue. To specify the stream for later reference, we supply the method with a string which will equal to the atom by which the queue is referred to as a stream in ECL^{i}PS^e. For example the following code creates two queues, one in each direction:

```
...
  ToEclipseQueue java_to_eclipse =
    eclipse.getToEclipseQueue("java_to_eclipse");
  FromEclipseQueue eclipse_to_java =
    eclipse.getFromEclipseQueue("eclipse_to_java");
...
```

These methods will create and return new *FromEclipseQueue* or *ToEclipseQueue* objects, and will also open streams with the specified names on the ECL^{i}PS^e side. No stream in ECL^{i}PS^e should exist with the specified name. If a stream exists which has this name and is not a queue between the Java object and ECL^{i}PS^e, the Java method throws an exception. If the name is used by a pre-existing queue, it is returned, so the `getFromEclipseQueue` and `getToEclipseQueue` methods can also be used to retrieve the queue objects by name once if they have already been created.

Opening a queue using ECL^{i}PS^e predicates

You can use the ECL^{i}PS^e builtin **peer_queue_create/5** to open a queue. Used correctly, these have the same effect as the Java methods explained above. For the peer name, you should use the atom returned by the `getPeerName()` method of the relevant *EclipseConnection* instance. The direction should be `fromec` for a *FromEclipseQueue* and `toec` for a *ToEclipseQueue*. The queue type should always be `sync` (for asynchronous queues, refer to section **??**).

Transferring data using Java methods

On the Java side, once a *FromEclipseQueue* has been established, you can treat it as you would any instance of *java.io.InputStream*, of which *FromEclipseQueue* is a subclass. Similarly, *ToEclipseQueue* is a subclass of *java.io.OutputStream*. The only visible difference is that *FromEclipseQueue* and *ToEclipseQueue* instances may have *QueueListeners* attached, as is discussed in Section 8.5.3.

Transferring data using ECL^{i}PS^e predicates

On the ECL^{i}PS^e side, there are built-in predicates for writing to, reading from and otherwise interacting with streams. Any of these may be used. Perhaps most useful are **read_exdr/2** and **write_exdr/2**; these are explained in Section 8.5.2. For the stream ID, you may either use the stream name, or the stream number, obtained for example using **peer_get_property/3**.

Note: always flush

When communicating between Java and ECL^{i}PS^e using queues, you should always invoke the `flush()` method of the Java *OutputStream* which you have written to, whether it be a *ToEclipseQueue* or an *EXDROutputStream*. Similarly, on the ECL^{i}PS^e side, `flush/1` should always be executed after writing. Although in some cases reading of the data is possible without a flush, flushing guarantees the transfer of data.

Closing a queue using Java methods

This is done simply by calling the `close()` method on the *FromEclipseQueue* or *ToEclipseQueue* instance.

Closing a queue using ECL^{i}PS^e predicates

This is done by executing the builtin **peer_queue_close/1**. Note that the builtin `close/1` should not be used in this situation, as it will not terminate the Java end of the queue.

8.5.2 Writing and reading ECL^{i}PS^e terms on queues

Rather than dealing with low-level data I/O instructions such as reading and writing bytes, the Java-ECL^{i}PS^e Interface provides classes for reading and writing whole terms. In the underlying implementation of these classes, the EXDR (ECL^{i}PS^e eXternal Data Representation) format is used. This allows ECL^{i}PS^e to communicate with other languages using a common data type. However, it is not necessary for the API user to know about EXDR in detail to use the Java-ECL^{i}PS^e Interface features discussed in this section.

EXDRInputStream is a subclass of *java.io.DataInputStream* which can read EXDR format. *EXDROutputStream* is a subclass of *java.io.FilterOutputStream* which can write EXDR format.

Initialising *EXDRInputStream* and *EXDROutputStream*

The constructor for *EXDRInputStream* takes an instance of *java.io.InputStream* as a parameter. This parameter stream is the source of the EXDR data for the new stream. If data has been written to the *InputStream* in EXDR format, you can access it by invoking the `readTerm` method of the new *EXDRInputStream*. This will read the data from the *InputStream* and translate the EXDR format into the Java representation of the data, which is then returned by `readTerm`.
Similarly, the constructor for *EXDROutputStream* takes an instance of *java.io.OutputStream* as a parameter. This parameter stream is the destination of the data written to the new stream. You write data by invoking the `write` method of the stream. The parameter of this method is a Java object representing the piece of data to be written. The class of this object can be any of the Java classes mentioned in Table 8.1. The object gets translated into EXDR format and this is written to the destination *OutputStream*.

EXDRInputStream and *EXDROutputStream* at work

Although the underlying stream could be any kind of stream (e.g. a file stream), the most common use of *EXDRInputStream* and *EXDROutputStream* is to read data from and write data to queues in EXDR format. In other words, we usually wrap these classes around *FromEclipseQueue* and *ToEclipseQueue* classes. We now look at an example which does just this.
The example is in these two files:

```
<eclipse_dir>/doc/examples/JavaInterface/QueueExample1.java
<eclipse_dir>/doc/examples/JavaInterface/queue_example_1.pl
```

The Java program's first relevant action is to invoke the `compile` method of the *EclipseEngine*. This causes the ECL^{i}PS^e program to be loaded by ECL^{i}PS^e engine. After `compile` completes, the Java program creates a *ToEclipseQueue* and a *FromEclipseQueue*, with the following lines:

```
// Create the two queues
java_to_eclipse = eclipse.getToEclipseQueue("java_to_eclipse");
eclipse_to_java = eclipse.getFromEclipseQueue("eclipse_to_java");
```

Then in the next two lines we create an *EXDROutputStream* to format data going to `java_to_eclipse` and an *EXDRInputStream* to format data coming from `eclipse_to_java`.

```
// Set up the two formatting streams
java_to_eclipse_formatted = new EXDROutputStream(java_to_eclipse);
eclipse_to_java_formatted = new EXDRInputStream(eclipse_to_java);
```

The Java program writes two atoms to `java_to_eclipse_formatted`, and then flushes the stream. This causes each atom to be translated into EXDR format and the translation to then be written on to `java_to_eclipse`. The Java program then makes an `rpc` invocation to the ECL^{i}PS^e program's only predicate `read_2_write_1/0`, which is defined as follows:

```
read_2_write_1:-
        read_exdr(java_to_eclipse, Term1),
```

```
read_exdr(java_to_eclipse, Term2),
write_exdr(eclipse_to_java, pair(Term1, Term2)),
flush(eclipse_to_java).
```

The built-in **read_exdr/2** reads a term's worth of data from the stream supplied and instantiates it to the second argument. So `read_2_write_1/0` reads the two terms from the stream. They are then written on to the `eclipse_to_java` stream within a `pair(...)` functor using the built-in **write_exdr/2**, and the stream is flushed. When the predicate succeeds, the `rpc` invocation returns and the term data is on `eclipse_to_java` in EXDR format. The next step of the java program is the following:

```
System.out.println(eclipse_to_java_formatted.readTerm());
```

Since `eclipse_to_java` was the *FromEclipseQueue* passed as a parameter when `eclipse_to_java_formatted` was initialised, the `readTerm` method of this object reads the EXDR data which is on `eclipse_to_java` and converts it into the appropriate Object to represent the piece of data, in this case a `CompoundTerm`. This Object is then returned by `readTerm`. Hence the output of the program is `pair(a,b)`.

8.5.3 Using the *QueueListener* interface

It may sometimes be useful to have Java react automatically to data arriving on a queue from ECL^{i}PS^e. An example of this would be where a Java program has a graphical display monitoring the state of search in ECL^{i}PS^e. We would like ECL^{i}PS^e to be able to send a message along a queue every time an element of the search state updates, and have Java react with some appropriate graphical action according to the message.
Similarly, ECL^{i}PS^e may require information from a Java database at some point during its operation. Again we could use a queue to transfer this information. If ECL^{i}PS^e tries to read from this queue when it is empty, we would like Java to step in and supply the next piece of data.
The *QueueListener* interface is the means by which handlers are attached to queues on the Java side so that Java reacts automatically to ECL^{i}PS^e's interaction with the queue.
Any object which implements the *QueueListener* interface can be attached to either a *FromEclipseQueue* or a *ToEclipseQueue*, using the `setListener` method. The *QueueListener* can be removed using `removeListener`. Queues can only have one Java listener at any one time.
The *QueueListener* interface has two methods: `dataAvailable` and `dataRequest`.

`dataAvailable` is invoked only if the *QueueListener* is attached to a *FromEclipseQueue*. It is invoked when the queue is flushed on the ECL^{i}PS^e side.

`dataRequest` is invoked only if the *QueueListener* is attached to a *ToEclipseQueue*. It is invoked when ECL^{i}PS^e tries to read from the queue when it is empty[1].

Both methods have a single *Object* parameter named `source`. When they are invoked this parameter is the *FromEclipseQueue* or *ToEclipseQueue* on which the flush or read happened.

[1]Note that this invocation occurs only if the ECL^{i}PS^e side of the queue is empty. If you have written data to the queue on the Java side, but not flushed it, the ECL^{i}PS^e side may still be empty, in which case the `dataRequest` method of the listener will be invoked.

There is an example Java program `QueueExample2.java` with an accompanying example ECL^{i}PS^e program `queue_example_2.pl` which use *QueueListeners* attached to queues going in both directions.

```
<eclipse_dir>/doc/examples/JavaInterface/QueueExample2.java
<eclipse_dir>/doc/examples/JavaInterface/queue_example_2.pl
```

After the queues streams are set up on both sides, the Java program attaches as listeners a *TermProducer* to the *ToEclipseQueue* and a *TermConsumer* to the *FromEclipseQueue*. These are both locally defined classes which implement *QueueListener*. The *TermProducer*, each time its `dataRequest` method is invoked, sends one of five different atoms down its queue in EXDR format. The *TermConsumer*, when its `dataAvailable` method is invoked, reads some EXDR data from its queue and translates it into the appropriate Java object. It then writes this object out to stdout.
Next, the Java program, using `rpc`, executes the only predicate in the ECL^{i}PS^e program: `read_5_write_5/0`. This repeats the following operation five times: read in a term in EXDR format from the relevant incoming stream, write it out in EXDR format with an extra functor to the relevant outgoing stream, and flush the outgoing stream.

8.5.4 Access to ECL^{i}PS^e's standard streams

If the object representing the ECL^{i}PS^e implements the *EclipseEngine* interface, then the API user may have access to the ECL^{i}PS^e's standard streams (see Section 8.7.2). These are returned as *FromEclipseQueue*s and *ToEclipseQueue*s by the methods `getEclipseStdin`, `getEclipseStdout` and `getEclipseStderr`.

8.6 Asynchronous Communicating between Java and ECL^{i}PS^e

Starting with release 5.9, the *OutOfProcessEclipse* and *RemoteEclipse* variants of the Java-ECL^{i}PS^e Interface also support asynchronous queues through the class *AsyncEclipseQueue*. These are essentially socket connections between ECL^{i}PS^e and Java, which can be used independently of which side has control.
An *AsyncEclipseQueue* queue is opened and closed in the same way as a *FromEclipseQueue* or *ToEclipseQueue*, but the following differences exist:

- an asynchronous queue can be read/written from the Java side even while the ECLiPSe side has control, e.g. during an RPC. This obviously has to happen from a different thread than the one that executes the RPC.
- I/O operations on asynchronous queues can block, they should therefore be done from a dedicated thread.
- the AsyncEclipseQueue class does not extend InputStream or OutputStream and can therefore not be used for I/O directly. Instead, a standard Java InputStream can be obtained from it via the getInputStream() method, and an OutputStream via the getOutputStream() method.
- on the ECLiPSe side, an event can (and should) be raised when data arrives from the Java side. If the ECLiPSe side has control at that time, it will handle the event. Otherwise, the event handling may be deferred until ECLiPSe has control back.

8.6.1 Opening and closing asynchronous queues

Opening and closing can be performed in a single step from either the Java or the ECL^{i}PS^e side.

Opening an asychronous queue using Java methods

The *AsyncEclipseQueue* does not have public constructors. Instead, we invoke `getAsyncEclipseQueue`. This asks the *EclipseConnection* object for a reference to an *AsyncEclipseQueue* instance which represents a new queue. To specify the stream for later reference, we supply the method with a string which will equal to the atom by which the queue is referred to as a stream in ECL^{i}PS^e. For example the following code creates such a queue:

```
 ...
  AsyncEclipseQueue java_eclipse_async =
    eclipse.getAsyncEclipseQueue("java_eclipse_async");
...
```

Note that this method will only work when the `eclipse` object is an *OutOfProcessEclipse* or *RemoteEclipse*. Teh method will create and return a new *AsyncEclipseQueue* object, and will also open a stream with the specified names on the ECL^{i}PS^e side. No stream in ECL^{i}PS^e should exist with the specified name. If a stream exists which has this name and is not a queue between the Java object and ECL^{i}PS^e, the Java method throws an exception. If the name is used by a pre-existing queue, it is returned, so the `getAsyncEclipseQueue` methods can also be used to retrieve the queue objects by name once they have already been created.

Opening an asychronous queue using ECL^{i}PS^e predicates

You can use the ECL^{i}PS^e builtin **peer_queue_create/5** to open a queue. Used correctly, these have the same effect as the Java methods explained above. For the peer name, you should use the atom returned by the `getPeerName()` method of the relevant *EclipseConnection* instance. The direction should be `bidirect`, and the queue type should be specified as `async`.

Transferring data using Java methods

Once an *AsyncEclipseQueue* has been created, a standard Java InputStream can be obtained from it via the getInputStream() method, and an OutputStream via the getOutputStream() method. These Java streams can be used as you would any instance of *java.io.InputStream* or *java.io.OutputStream*. Unlike the synchronous *FromEclipseQueue* and *ToEclipseQueue*, I/O on these streams can block, and should therefore be handled by dedicated threads. For this reason, the listener-feature is not supported on asynchronous queues.

Transferring data using ECL^{i}PS^e predicates

On the ECL^{i}PS^e side, there are built-in predicates for writing to, reading from and otherwise interacting with streams. Any of these may be used. Perhaps most useful are **read_exdr/2** and **write_exdr/2**; these are explained in Section 8.5.2. For the stream ID, you may either use the stream name, or the stream number, obtained for example using **peer_get_property/3**.
Since the ECL^{i}PS^e side does not support threads, it should only write to an asynchronous queue when there is an active thread on the Java side to read the data. Otherwise, the write-operation may block, and control will never be transferred back from ECL^{i}PS^e to Java.

For reading from an asynchronous queue, the ECL^{i}PS^e side should typically set up an event handler. The handler will be invoked whenever new data arrives on a previously empty queue.

```
my_async_queue_handler(Stream) :-
    ( select([Stream], 0, [_]) ->
        read_exdr(Stream, Data),
        process(Data),
        my_async_queue_handler(Stream)
    ;
        events_nodefer
    ).

setup_async_queue_with_handler :-
    event_create(my_async_queue_handler(my_async_queue), [defers], Event),
    peer_queue_create(my_async_queue, host, async, bidirect, Event).
```

Note that execution of the handler is only guaranteed when the ECL^{i}PS^e side has control. When communicating between Java and ECL^{i}PS^e using queues, you should always invoke the `flush()` method of the Java *OutputStream* which you have written to, whether it be a *ToEclipseQueue* or an *EXDROutputStream*. Similarly, on the ECL^{i}PS^e side, `flush/1` should always be executed after writing. Although in some cases reading of the data is possible without a flush, flushing guarantees the transfer of data.

Closing a queue using Java methods

This is done simply by calling the `close()` method on the *AsyncEclipseQueue* instance.

Closing a queue using ECL^{i}PS^e predicates

This is done by executing the builtin **peer_queue_close/1**. Note that the builtin `close/1` should not be used in this situation, as it will not terminate the Java end of the queue.

8.6.2 Writing and reading ECL^{i}PS^e terms on queues

See the corresponding section for synchronous queues, 8.5.2.

8.7 Managing connections to ECL^{i}PS^e

As of ECL^{i}PS^e 5.3 and later, there are three Java classes which can be used to provide interaction with ECL^{i}PS^e. These are *EmbeddedEclipse*, *OutOfProcessEclipse* and *RemoteEclipse*. Although the three classes all implement the *EclipseConnection* interface, there are some important differences in initialisation, termination and use which are explained in this section. On the ECL^{i}PS^e side, as of version 5.3 and later, there is a unified interface for interaction with Java, based around the concept of a 'peer'. This is explained in Section 8.7.1. Section 8.7.2 discusses how to create an ECL^{i}PS^e engine from within Java (this covers both *EmbeddedEclipse* and *OutOfProcessEclipse*). Section 8.7.3 discusses the *RemoteEclipse* class, which allows Java to connect to an existing ECL^{i}PS^e engine. Finally, Section 8.7.4 compares the three connection classes, assessing the advantages and disadvantages of each.

8.7.1 A unified ECL^{i}PS^e-side interface to Java : the 'peer' concept

To allow ECL^{i}PS^e code to be independent of the way it is interfacing with Java, there is a unified technique in ECL^{i}PS^e for interfacing to Java and other non-ECL^{i}PS^e programs, based on the concept of a *peer*. A peer is a computation environment which is external to ECL^{i}PS^e, either in the sense of being a different computer language or being a different process, possibly on a different machine. When ECL^{i}PS^e is communicating with one or more Java virtual machines using the Java-ECL^{i}PS^e interface (including cases where ECL^{i}PS^e is embedded), it stores some global information to keep track of each peer.
Each peer of the ECL^{i}PS^e engine is indexed within ECL^{i}PS^e by a unique *peer name* which is an atom. This is set during the initialisation of the connection between ECL^{i}PS^e and Java; the details vary according to the Java class used to make the connection.
The peer name is used in those ECL^{i}PS^e predicate calls related to the peer connection. In particular, those related to the opening and closing of peer queues (see Section 8.5.1). The *EclipseConnection* interface includes the method `getPeerName()` which can be used to retrieve this name on the Java side.

8.7.2 Creating and managing ECL^{i}PS^e engines from Java

There are at present two options for creating an ECL^{i}PS^e engine from Java. With *EmbeddedEclipse*, the engine takes the form of a dynamically loaded shared library within the Java virtual machine. With *OutOfProcessEclipse* it is a separate, child process of the Java virtual machine. These two options have in common that they are both initialised using an *EclipseEngineOptions* object and that both classes implement *EclipseEngine* .

Configuring an *EclipseEngineOptions* object

Before an ECL^{i}PS^e engine is created using either option, an *EclipseEngineOptions* object must be created and configured. An instance of the *EclipseEngineOptions* class represents our configuration choices for a new ECL^{i}PS^e engine.
The options can be specified by either looking them up in a *java.util.Properties* instance or by calling "set" methods on the *EclipseEngineOptions* instance.
In the first case you can either specify system properties (by passing `-D` command line options to the JVM) or you can use an *EclipseEngineOptions* constructor which takes a *Properties* parameter. Each option has a standard property name detailed below.
Once the *EclipseEngineOptions* object is created, there are also "set" methods we can invoke on it to set the different options.

The ECL^{i}PS^e installation directory (`eclipse.directory`) This must be the correct path of an ECL^{i}PS^e installation so that Java can find the files it needs to start ECL^{i}PS^e. If an *EclipseEngineOptions* constructor is invoked without a directory having been set, either using a constructor parameter or as a property, an exception will be thrown.

The default module (`eclipse.default-module`) is the default ECL^{i}PS^e module where goals are executed. If no setting is made, then the default module will be "eclipse".

Local and global size (`eclipse.local-size` and `eclipse.global-size`) are the maximum size of the local stack and the global stack respectively. In each case the option

is an integer representing the required size in megabytes. If no setting is made, the sizes default to the ECL^{i}PS^e defaults.

Local size is the maximum combined size of the local stack and the control stack. Roughly speaking, the local stack is the ECL^{i}PS^e equivalent of the call/return stack in procedural languages. The control stack is used to store the choicepoints which occur in the execution of non-deterministic code. You may therefore need to increase this limit if your search tree is exceptionally deep or if there is a lot of non-determinism. You can use the ECL^{i}PS^e goal `get_flag(max_local_control, X)` to find out the current setting. For more details, see the section on "Memory Organisation and Garbage Collection" in the ECL^{i}PS^e User Manual.

Global size is the maximum combined size of the global stack and the trail stack. The global stack is used to store data structures. The trail stack is used to store backtracking information and is therefore closely related to the control stack. You may need to increase this limit if you are creating large data structures in ECL^{i}PS^e, or if there is a lot of non-determinism. You can use the ECL^{i}PS^e goal `get_flag(max_global_trail, X)` to find out the current setting. Again, see the section in the ECL^{i}PS^e User Manual for more details.

The "use queues" flag (`eclipse.use-queues`) If `false` (the default case), the ECL^{i}PS^e engine's standard streams (`stdin`, `stdout` and `stderr`) will be connected to the standard streams of the JVM. So for example if, as in `QuickTest.java`, a message is written to `stdout` from within ECL^{i}PS^e, this will appear on the `stdout` stream of the JVM. Similarly, if ECL^{i}PS^e requests input from the `stdin` stream, input will be requested from the `stdin` stream of the JVM. If the flag is set `true`, *FromEclipseQueue* and *ToEclipseQueue* objects are used to represent the ECL^{i}PS^e's standard streams.

The peer name (`eclipse.peer-name`) is the peer name by which the Java side can be referenced within ECL^{i}PS^e, for example when queues are created from the ECL^{i}PS^e side. By default the peer name for the Java side when using an *EmbeddedEclipse* or an *OutOfProcessEclipse* is "host".

Using *EmbeddedEclipse*

This section discusses issues specific to using the *EmbeddedEclipse* class. With *EmbeddedEclipse*, the ECL^{i}PS^e engine is a dynamically loaded native library in the Java virtual machine. Figure 8.1 shows this deployment model in UML notation. The important consequences of this deployment model are:

- The ECL^{i}PS^e engine shares memory and other resources with Java.
- Only one such engine can exist in a Java virtual machine at any one time.
- Communication between Java and ECL^{i}PS^e is very efficient.

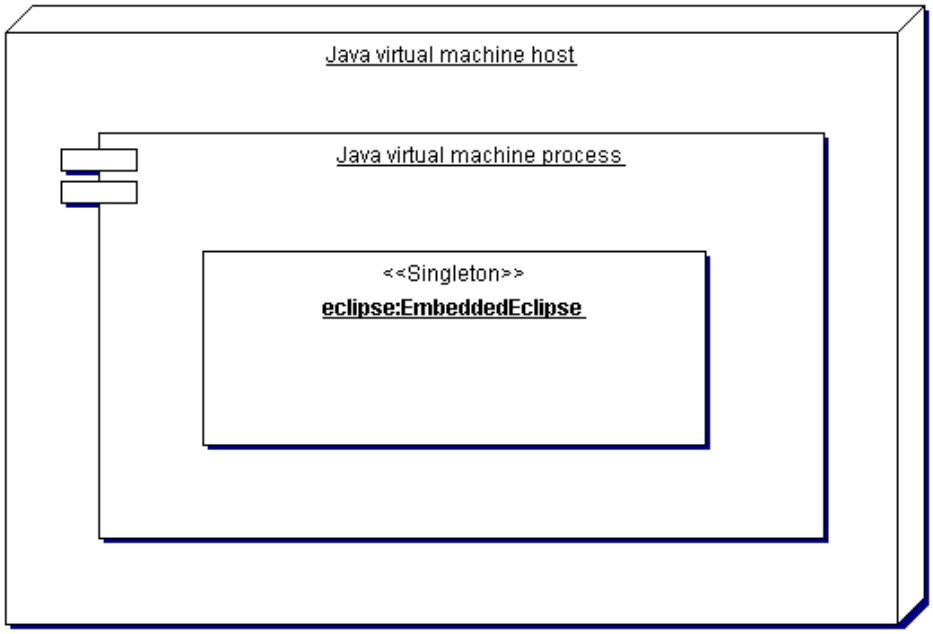


Figure 8.1: UML deployment diagram for *EmbeddedEclipse*

Initialising an *EmbeddedEclipse* The embedded ECL^{i}PS^e which runs within the JVM and shares its resources, can be started and ended only once during the lifetime of the JVM. There is no public constructor method for *EmbeddedEclipse*. Initialisation of the embedded ECL^{i}PS^e is done using the static `getInstance` method of class *EmbeddedEclipse* which takes an *EclipseEngineOptions* instance as a parameter. The method uses this to configure and set up ECL^{i}PS^e and then returns an object of type *EmbeddedEclipse*. There may only ever be one instance of *EmbeddedEclipse* in a JVM. If the embedded ECL^{i}PS^e has already been set up or if it has been set up and terminated, subsequent invocations of `getEclipse` with an *EclipseEngineOptions* will throw exceptions. However during the lifetime of the embedded ECL^{i}PS^e, a reference to the unique *EmbeddedEclipse* object can be obtained using the parameterless static `getEclipse` method.

Termination of an *EmbeddedEclipse* The `destroy` method which appears in the *EmbeddedEclipse* class will shut the embedded ECL^{i}PS^e down. Once the `destroy` method has been invoked, the invocation of any methods which require use of the ECL^{i}PS^e engine will result in an *EclipseTerminatedException* being thrown. The `destroy` method should free all the resources of the JVM process which were being used by the embedded ECL^{i}PS^e.

Once the *EmbeddedEclipse* has been destroyed, `getEclipse` can no longer be used during the lifetime of the JVM to initialise an embedded ECL^{i}PS^e engine. In other words, by invoking `destroy`, one removes the ability to use embedded ECL^{i}PS^e engines within the current instance of the JVM.

Using *OutOfProcessEclipse*

This section discusses issues specific to the *OutOfProcessEclipse* class. With *OutOfProcessEclipse*, the ECL^{i}PS^e engine is a child process of the Java virtual machine. Figure 8.2 shows this deployment model in UML notation. The important consequences of this deployment model are:

- The ECL^{i}PS^e engine uses separate memory and other resources, depending on how the operating system allocates these between processes.
- Several instances of *OutOfProcessEclipse* can exist in a Java virtual machine at any one time.
- Communication between Java and ECL^{i}PS^e is less efficient.

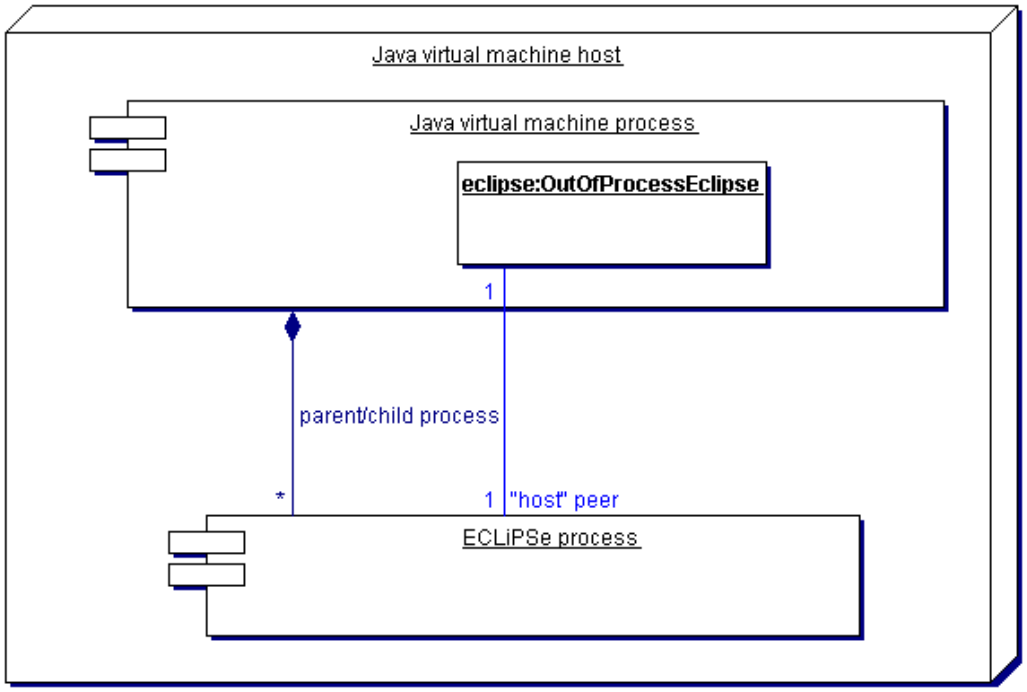


Figure 8.2: UML deployment diagram for *OutOfProcessEclipse*

Initialisation of an *OutOfProcessEclipse* *OutOfProcessEclipse* has a single constructor which takes an *EclipseEngineOptions* object as its only parameter. See Section 8.7.2 for details of how to create and configure this object. Unlike *EmbeddedEclipse*, multiple *OutOfProcessEclipse* instances are allowed.

Termination of an *OutOfProcessEclipse* We invoke the instance method `destroy()` in *OutOfProcessEclipse* to terminate both the child ECL^{i}PS^e process and our association with it. Once the `destroy` method has been invoked, the invocation of any methods on the destroyed

OutOfProcessEclipse object which require use of the ECL^{i}PS^e engine will throw an *EclipseTerminatedException*. Unlike *EmbeddedEclipse*, invoking `destroy()` on an *OutOfProcessEclipse* does not affect our ability to create new *OutOfProcessEclipse* instances during the lifetime of the Java virtual machine.
If the child process ECL^{i}PS^e crashes or is killed while ECL^{i}PS^e has control, the Java thread which handed control to ECL^{i}PS^e should throw an *EclipseTerminatedException*. If this happens while Java has control, usually the next invocation of a method on the *OutOfProcesEclipse* should throw an *EclipseTerminatedException*, although it is possible that some operations will throw a different class of *IOException*. If this should happen it is worth calling the `destroy` method to do a final clean-up.

8.7.3 Connecting to an existing ECL^{i}PS^e engine using *RemoteEclipse*

In some applications, for example where Java is used to visualise search in ECL^{i}PS^e, the life of the ECL^{i}PS^e engine may begin before the connection with Java is initialised or end after the connection with Java is terminated. Furthermore, it may also be useful for the eclipse engine and the Java virtual machine to be running on physically separate computers, for example if the ECL^{i}PS^e tasks are being executed on a compute server, but the Java program is to be run on a workstation. The *RemoteEclipse* class can be used to connect Java to ECL^{i}PS^e in these two scenarios. The deployment model is that the *RemoteEclipse* Java object is a "Proxy" for the ECL^{i}PS^e engine which is running on the remote machine, as shown in UML notation in Figure 8.3.

The key consequences of this deployment model are:

- ECL^{i}PS^e and Java can run on different machines and the two machines may have a different architecture/OS.
- ECL^{i}PS^e can connect to multiple Java processes using this model.
- The lifetime of the ECL^{i}PS^e engine need not necessarily be a sub-duration of the lifetime of the JVM.

Initialisation of a *RemoteEclipse* connection

Connecting Java to ECL^{i}PS^e using *RemoteEclipse* requires the ECL^{i}PS^e engine to be primed so that it is ready to accept the connection. By the time it connects, the Java program must have the IP address of the machine hosting the ECL^{i}PS^e engine (the server) and the port number being used for the connection. The attachment protocol also optionally allows for a password to be used by the Java side and checked against one specified on the ECL^{i}PS^e side. Also the server must be configured to allow TCP/IP socket servers which can be connected to by the machine hosting Java. Initialising a connection using *RemoteEclipse* therefore requires some coordination between the ECL^{i}PS^e code and the Java code. The Java code always consists of a single *RemoteEclipse* constructor invocation, although the constructor parameters may vary.
The ECL^{i}PS^e side of the code uses certain builtins. Refer to the relevant documentation of these for precise details of usage. On the ECL^{i}PS^e side, the code can be structured in two different ways; one simpler and the other allowing more flexibility. We outline here the sequence of actions in each case.

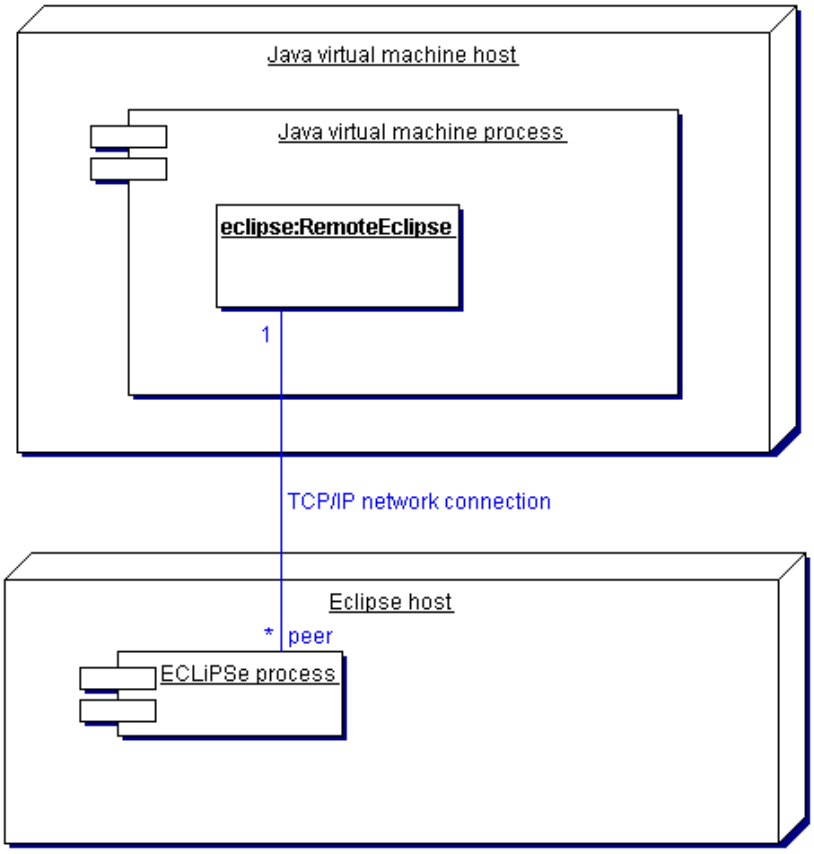


Figure 8.3: UML deployment diagram for *RemoteEclipse*

Basic connection sequence This can be used in situations where no password is to be used and where the port number is specified in advance, rather than generated dynamically by ECL^{i}PS^e, and is known by the Java side.

1. The ECL^{i}PS^e side executes the builtin **remote_connect/3**, specifying the port number in advance. This builtin will block until the connection is established.

2. The Java side then invokes one of the *RemoteEclipse* constructors which has no password parameter. This should immediately complete or throw an exception if the connection is unsuccessful.

Advanced connection sequence This more complicated sequence uses a password and optionally allows the port number to be generated dynamically and then communicated to the Java side.

1. The ECL^{i}PS^e side executes the builtin **remote_connect_setup/3**, specifying the password and either specifying the port number or allowing it to be generated dynamically.

2. The port number must be communicated to the Java side somehow, e.g. manually, or via a file.

3. The Java side then invokes one of the *RemoteEclipse* constructors with a password parameter. This either blocks until the connection is successful or throws an exception.

4. The ECL^{i}PS^e side executes the builtin **remote_connect_accept/6**, specifying the password which the Java should supply.

5. The Java constructor invocation then completes, or throws an exception if the connection could not be made.

If left as a free variable, the `Host` argument of either the **remote_connect/3** or **remote_connect_setup/3** goal will become instantiated to the IP address of the machine hosting ECL^{i}PS^e. Another possibility is to call the goal with this argument already instantiated to the atom `localhost`. This will mean that only client connections made by processes on the same machine and using the loopback address will be accepted. With this usage, on the Java side you should use invoke `InetAddress.getHostByName("localhost")`. Note that `InetAddress.getLocalHost()` will not work in this situation.

In both connection sequences, the peer name indexing the connection is either specified or generated dynamically on the ECL^{i}PS^e side in the **remote_connect/3** or **remote_connect_setup/3** goal.

Once the connection has been established by one of the above sequences, control initially rests with the Java side. Therefore the ECL^{i}PS^e code which called the **remote_connect/3** goal or the **remote_connect_accept/6** goal blocks until the Java side explicitly transfers control to ECL^{i}PS^e or disconnects.

Explicit transfer of control between ECL^{i}PS^e and Java

As mentioned above, after the initial connection has been established, Java has control by default. However, this may not be convenient. For example, in the case of search visualisation, after the initialisation of the visualisation client, we may prefer ECL^{i}PS^e to have control by default, allowing control to pass to Java only on certain occasions. Control can be explicitly passed from Java to ECL^{i}PS^e by invoking the `resume()` method on a *RemoteEclipse*. In this case the ECL^{i}PS^e code will resume execution after the last point where it passed control to Java. For example, if `resume()` is invoked immediately after the *RemoteEclipse* constructor completes, ECL^{i}PS^e execution will resume at the point just after the call to the **remote_connect/3** goal or the **remote_connect_accept/6** goal.

Control can be transferred to a Java peer using the **remote_yield/1** builtin. In this case the Java thread which passed execution to ECL^{i}PS^e will resume execution at the point where it blocked.

The `resume()` method and the **remote_yield/1** builtin should be used with care. An invocation of `resume()` should be paired with an execution of a **remote_yield/1** goal in most cases. In addition, **remote_yield/1** should not be executed in any code executed as a result of an `rpc` invocation and `resume()` should not be executed within the *QueueListener* methods `dataAvailable()` or `dataRequest()`.

The `resume()` method and the **remote_yield/1** builtin should only be used when other techniques such as `rpc` are not suitable.

Termination of a *RemoteEclipse* connection

A *RemoteEclipse* connection between ECL^{i}PS^e and Java may be terminated in different ways. Firstly, disconnection may be initiated by either side. Secondly the disconnection may be either *multilateral* or *unilateral*. Multilateral disconnection, the preferred method, is where the side which has control initiates disconnection. Unilateral disconnection is where the side which initiates disconnection does not have control, and should only take place as a clean-up routine when one side is forced to terminate the connection because of an unexpected event.

Java-initiated multilateral disconnect is performed by invoking the `disconnect` method on the *RemoteEclipse* instance while the Java side has control.

Java-initiated unilateral disconnect is performed by invoking the `unilateralDisconnect` method on the *RemoteEclipse* instance while Java does not have control.

ECL^{i}PS^e-initiated multilateral disconnect is performed on the ECL^{i}PS^e side by executing a **remote_disconnect/1** goal while ECL^{i}PS^e has control, identifying the connection to be closed by supplying the peer name.

ECL^{i}PS^e-initiated unilateral disconnect cannot be executed by the user, since ECL^{i}PS^e is not multi-threaded. However, it may occur in certain urgent situations e.g. if the ECL^{i}PS^e process is killed while ECL^{i}PS^e does not have control.

If an ECL^{i}PS^e-initiated disconnect occurs, or the connection is lost for whatever reason while ECL^{i}PS^e has control, the Java thread which handed control to ECL^{i}PS^e should throw an *EclipseTerminatedException*. If either of these happens while Java has control, the next invocation of a method on the *RemoteEclipse* should throw an *EclipseTerminatedException*, although it is possible that some operations will throw a different class of *IOException*. If this should happen it is worth invoking the `unilateralDisconnect()` method to do a final clean-up.

8.7.4 Comparison of different Java-ECL^{i}PS^e connection techniques

This section should give you some idea of when it is most appropriate to use each connection class. Figure 8.4 is a UML class diagram showing the most important relationships and operations of the principal classes and interfaces.
All three classes implement *EclipseConnection* , which provides all the functionality you would expect during a "session" with ECL^{i}PS^e. The *EclipseEngine* interface is implemented when the JVM "owns" the ECL^{i}PS^e engine, and so provides the methods to access the standard I/O streams. Note that the termination methods are not in either of the interfaces, but are specific to each class. Furthermore, the `resume()` method allows *RemoteEclipse* to explicitly hand control to ECL^{i}PS^e, but this operation is not supported by the other two classes.
To summarise the advantages and disadvantages Table 8.2 gives an at-a-glance comparison of the different features of the different connection classes.

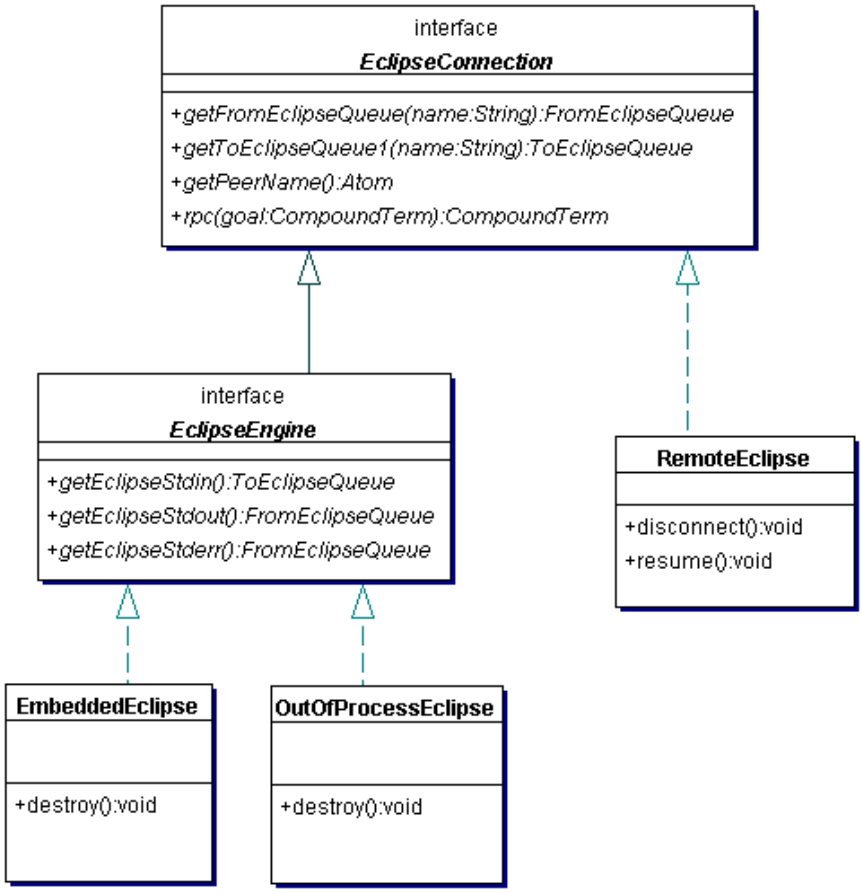


Figure 8.4: UML class diagram for different classes connecting Java and ECL^{i}PS^e. Some convenience methods from *EclipseConnection* have been omitted.

Feature	Java-ECL^{i}PS^e connection class		
	Embedded	*OutOfProcess*	*Remote*
Implements *EclipseConnection* interface (allowing `rpc` and queues)	•	•	•
Implements *EclipseEngine* interface (allowing access to ECL^{i}PS^e stdio streams)	•	•	–
ECL^{i}PS^e is in a separate process (with separate memory heap/stack)	–	•	•
ECL^{i}PS^e can be on a separate machine from Java	–	–	•
ECL^{i}PS^e engine can start before/ end after Java virtual machine	–	–	•
ECL^{i}PS^e engine created/ destroyed from Java	•	•	–
Efficient transfer of data on queues and `rpc` invocations	•	–	–
One ECL^{i}PS^e can connect to many Java virtual machines using this	–	–	•
One Java virtual machine can connect to many ECL^{i}PS^e engines using this	–	•	•

Table 8.2: Feature comparison table for different ECL^{i}PS^e connection classes

Chapter 9

EXDR Data Interchange Format

We have defined a data interchange format called EXDR for the communication between ECL^{i}PS^e and other languages. The data types available in this format are integer, double, string, list, nil, structure and anonymous variable. This is intended to be the subset of ECL^{i}PS^e types that has a meaningful mapping to many other languages' data types. The mapping onto different languages is given in the following table. For details of the mapping between Java classes/interfaces and EXDR/ECL^{i}PS^e types see Section 8.3.1.

EXDR type	**ECLiPSe type**	**TCL type**	**Java type**
Integer	integer	int	java.lang.Integer
e.g.	123	123	123
Long	integer	string	java.lang.Long
e.g.	5000000000	5000000000	5000000000
Double	float	double	java.lang.Double/Float
e.g.	12.3	12.3	12.3
String	string	string	java.lang.String
e.g.	”abc”	abc	”abc”
List	./2	list	java.util.Collection
e.g.	[a,b,c]	{a b c}	
Nil	[ ]/0	empty string	java.util.Collection
e.g.	[ ]	{} ””	
Struct	compound	list	CompoundTerm/Atom
e.g.	foo(bar,3)	{foo bar 3}	
Variable	variable	string	null
e.g.	_	_	

The EXDR Integer data type is a 32-bit signed integer, the EXDR Long data type is a 64-bit signed integer, bigger ECL^{i}PS^e integers cannot be represented. The EXDR Variable type only allows singleton, anonymous variables, which means that it is not possible to construct a term where a variable occurs in several places simultaneously. The main use of these variables is as placeholders for result arguments in remote procedure calls.

9.1 ECL^{i}PS^e primitives to read/write EXDR terms

The ECL^{i}PS^e predicates to create and interpret EXDR-representation read from and write directly to ECL^{i}PS^e streams. This means that EXDR-format can be used readily to communicate via files, pipes, sockets, queues etc.

write_exdr(+Stream, +Term)
This predicate writes terms in exdr format. The type of the generated EXDR-term is the type resulting from the "natural" mapping of the Eclipse terms. Atoms are written as structures of arity 0 (not as strings). Note that all information about variable sharing, variable names and variable attributes is lost in the EXDR representation.

read_exdr(+Stream, -Term)
This predicate reads exdr format and constructs a corresponding Eclipse term.

Please refer to chapter 5 for the Tcl primitives, and to chapter 8 for the Java primitives for manipulating EXDR terms.

9.2 Serialized representation of EXDR terms

The following is the specification of what is actually send over the communication channels. This is all the information needed to create new language mappings for EXDR terms. This definition corresponds to EXDR_VERSION 2:

```
ExdrTerm      ::=   'V' Version CompactFlag? Term
CompactFlag   ::=   'C'
Term          ::=   (Integer|Double|String|List|Nil|Struct|Variable)
Integer       ::=   ('B' <byte> | 'I' XDR_int | 'J' XDR_long)
Double        ::=   'D' XDR_double
String        ::=   ('S' Length <byte>* | 'R' Index)
List          ::=   '[' Term (List|Nil)
Nil           ::=   ']'
Struct        ::=   'F' Arity String Term*
Variable      ::=   '_'
Length        ::=   XDR_nat
Index         ::=   XDR_nat
Arity         ::=   XDR_nat
Version       ::=   <byte>
XDR_int       ::=   <4 bytes, msb first>
XDR_long      ::=   <8 bytes, msb first>
XDR_double    ::=   <8 bytes, ieee double, exponent first>
XDR_nat       ::=   ( <8 bits: 1 + seven bits unsigned value>
                    | XDR_int )                    // >= 0
```

The version byte is 1 or 2. EXDR version 1 encodings are also valid version 2 encodings, and version version 2 decoders can read version 1 encoded terms.
XDR_long, XDR_int and byte are all signed integers in two's complement representation.
The string reference code R means that the string is the same as the Index'th S-encoded string that occurred in the EXDR term earlier. The presence of the CompactFlag C in the header

indicates that the term may actually contain such string references. If the flag is absent, the term does not contain any.

Chapter 10

The Remote Interface Protocol

10.1 Introduction

The ECL^{i}PS^e remote interface protocol is used to build a remote interface between ECL^{i}PS^e and some programming language. A program written in that programming language can interact and communicate with a separate ECL^{i}PS^e process via the remote interface. The Tcl remote interface (chapter 6) is an example of such an interface. This chapter describes the protocol, so that remote interfaces to other programming languages can be built.
The protocol is designed to allow the implementer to build an interface that is compatible with the embedding interface of the same language. This should allow the same code (in both ECL^{i}PS^e and the other language) to be used in both interfaces. On the ECL^{i}PS^e side, the concept of *peers* is used to unify the remote and embedding interfaces.
Another feature of the remote interface is that on the ECL^{i}PS^e side, the interface is independent of the programming language that is being interfaced to. It should be possible to write ECL^{i}PS^e code with the interface (e.g. for a GUI) and change the remote code without needing to rewrite the code on the ECL^{i}PS^e side.
Briefly, a socket connection is established between the remote program and an ECL^{i}PS^e process. The processes exchange messages in the EXDR (see chapter 9) format according to the protocol. This allows the communication to be platform independent, and the ECL^{i}PS^e and remote processes can be located on any two machines which can establish socket connections.

10.2 Basics

The remote interface is established by **attaching** the remote and ECL^{i}PS^e processes. The attachment establishes two socket connections between the two processes:

Control This connection is used to control the remote interface. Messages (in EXDR format) are sent in both directions according to the remote protocol to co-ordinate the two processes.

Rpc This is used to send **ec_rpc** goals from the remote process to ECL^{i}PS^e and return the results. The goal is sent in EXDR format.

More than one remote attachment can be established in an ECL^{i}PS^e process. Each attachment is independent, and is a remote peer, identified by its control connection. Each remote attachment has two sides: the ECL^{i}PS^e side, and the remote side.

At any one time, either the ECL^{i}PS^e or the remote side has *control*. When a side has control, it is able to send messages to the other side via the control connection. The side that does not have control waits for messages to arrive on the control connection. On the ECL^{i}PS^e side, execution is suspended while it does not have control. In general, once a control message is sent, the control is passed to the other side, and the side that sent the message waits for a reply message from the other side.
The **ec_rpc** mechanism is designed to be the main way for the remote side to interact with the ECL^{i}PS^e side. The remote side can send an ECL^{i}PS^e goal, in EXDR format to be executed by the ECL^{i}PS^e side. This can only be done while the remote side has control, and when the goal is issued, a message is sent via the control connection to the ECL^{i}PS^e side, and control is passed to the ECL^{i}PS^e side. Control is passed back to the remote side when ECL^{i}PS^e completes the execution of the goal.
After the attachment, extra I/O connections can be established between the two sides. This allows data to be transferred from one side to the other. These connections (referred to as peer queues) can be of two types:

synchronous These queues are synchronised by the control connection. Control messages are exchanged between the two sides to ensure that they are both are synchronised for the data transfer: one side consumes the data that is sent from the other. This ensures that no blocking occurs with the I/O operations across the sockets.

asynchronous These queues can perform I/O operations that are not co-ordinated by the control connection. Either side can write to or read from the queue without transferring control. In fact, if the remote language is multi-threaded, it can perform asynchronous I/O while ECL^{i}PS^e side has control. Note that asynchronous I/O operations may block on the ECL^{i}PS^e side.

10.3 Attachment

10.3.1 Attachment Protocol

The attachment process is summarised in Figure 10.3. It is initiated from the ECL^{i}PS^e side, by either calling **remote_connect/3** or the more flexible **remote_connect_setup/3** and **remote_connect_accept/6** pair. In fact, **remote_connect/3** is implemented using **remote_connect_setup/3** and **remote_connect_accept/6** with default values for some of the arguments.
The attachment can be divided into several phases:

1. Initialisation and handshaking: the control connection is established, and handshaking between the two sides are carried out – checking that the remote protocols on the two sides are compatible; checking the pass-term.

2. Establishing the ERPC connections and exchange of information (remote language, ECL^{i}PS^e name for the peer).

3. User-defined initialisations.

The remote protocol version is stored in the flag `remote_protocol_version`, accessible via **get_flag/2** as an integer version number. This version number should only change when the

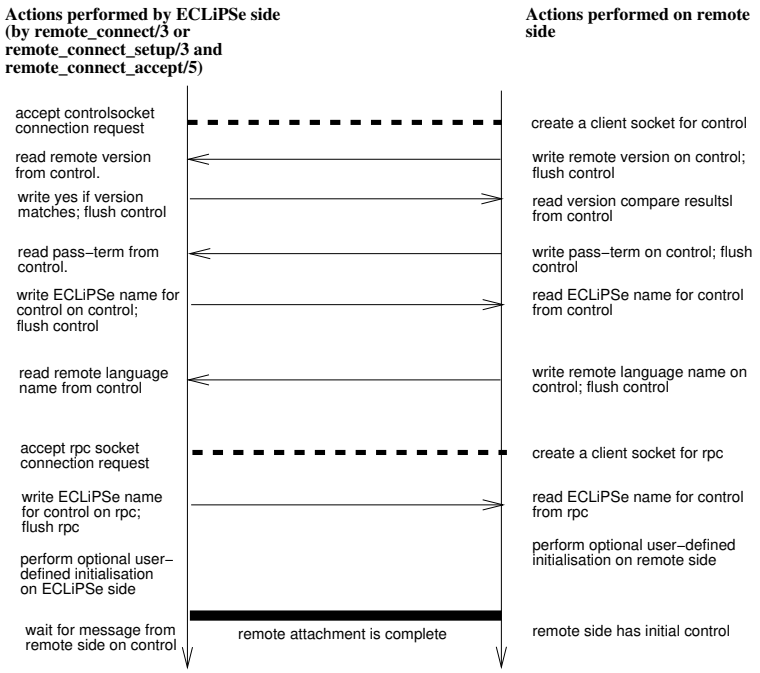


Figure 10.1: Summary of the attachment protocol

protocol is modified. Checking of the version ensures that the same (or at least compatible) versions of the protocol are used, so that the two sides behaves correctly. The version information is sent from the remote side (which must have its own copy of the version information), and the ECL^{i}PS^e side checks that this is compatible with the protocol version it is using. In order to cope with remote connections which may not be using the remote protocol, the ECL^{i}PS^e side waits only for a fixed period of time for the remote side to send the version information before timing out.
Time-out on the ECL^{i}PS^e side can also occur for forming any of the connections between the two sides, from the control connection to the peer queue. This is specified by the user in `remote_connect_accept/6`. If time-out occurs during the attachment, then the attachment process is abandoned, and the predicate fails (any connected sockets will be closed).
The detailed sequence of events for the attachment for the remote side (with some description of the relevant ECL^{i}PS^e side actions) are:

1. ECL^{i}PS^e side: a socket server for the control connection is created, using the address Host/Port which can be specified by the user. It then waits to accept a socket stream for the control connection from the remote side. The user can specify the amount of time to wait before this operation times-out, which would then terminate the attachment.

2. Remote side: create a client socket stream for the control connection, with an address compatible with Host/Port. The socket stream should be in blocking mode, and perform no translation on the data sent.

3. Remote side: sends the remote protocol version information on the control socket in EXDR format and flush it. This should be a `remote_protocol/1` term.

4. ECL^{i}PS^e side: reads the EXDR version term from the control socket, and compares with the version on the ECL^{i}PS^e side. If the two protocols are compatible, it sends the EXDR string `yes` back to the remote side; otherwise it sends the ECL^{i}PS^e remote protocol version to the remote side and disconnects from the remote side (raising unimplemented functionality error). The ECL^{i}PS^e side waits at most 100 seconds after the control connection is established for the remote side's version: after this the ECL^{i}PS^e side disconnects the control connection and raises out of range error.

5. It is expected that the future versions of the protocol will remain unchanged up to this point at least, to ensure the proper handshaking and checking of versions.

6. Remote side: write the 'pass-term' in EXDR format on the newly created control connection and flush it. This provides a simple security check: ECL^{i}PS^e side will check if the 'pass-term' matches the 'pass-term' it was given when the remote connection was initiated – for **remote_connect/3**, the pass-term is the empty string, but the user can specify any pass-term if **remote_connect_setup/3** and **remote_connect_accept/6** are used. If the terms are not identical, then the ECL^{i}PS^e side will discontinue the attachment process.

7. Remote side: read from the control connection the ECL^{i}PS^e name for the control connection. This is sent in EXDR format, and is used to identify this particular remote attachment – the peer name for the peer. This name is needed when calling ec_rpc goals that refer to the peer.

8. Remote side: write the name of the programming language (e.g. tcl, java) of the remote process on the control connection. This should be in EXDR string format, and the connection flushed.

9. ECL^{i}PS^e side: read the name of the programming language and store it (it can be accessed later via **peer_get_property/3**). The ECL^{i}PS^e side now wait to accept the socket stream (using the same socket server as the control connection) for the ec_rpc connection. This can also time-out.

10. Remote side: create a client socket stream for the ec_rpc connection, using the same Port as for the control connection. This stream should be in blocking mode, and perform no translation on the data sent. The server socket on the ECL^{i}PS^e will be closed after accepting this client.

11. Remote side: read the control connection name again on the remote side, on the newly established ec_rpc connection. This is also sent in EXDR format. This is designed to verify that the ec_rpc connection is indeed connected to the ECL^{i}PS^e side.

12. Remote side: the remote side now has control. Any user-defined initialisations on the remote side can now be performed, to make the remote side ready for the interaction. The remote side has the control initially. Note that user-defined initialisations on the ECL^{i}PS^e side is also performed after sending the control name on the ec_rpc connection. After the initialisation, ECL^{i}PS^e side will suspend and listen on the control connection for the remote side to give control back to the ECL^{i}PS^e side.

At the end of this, the remote side should be ready for normal interaction with the ECL^{i}PS^e side.

The remote_connect/3 or remote_connect_accept/6 predicate waits for the control to be handed back by the remote side before exiting. Thus when the predicate succeeds, the remote side has been attached and properly initialised, with ECL^{i}PS^e side having control.
The protocol does not specify how the remote side should be informed of the Host/Port address for the initial socket connection. The Address can be fixed before hand (with the Address argument instantiated, or the information can be transmitted either manually or via files. In addition, the remote process can be started from within ECL^{i}PS^e using the **exec/3** command, with the host and port supplied as arguments.
In accepting client socket connections from the remote side, the ECL^{i}PS^e side is informed of the host of the remote side. This should either be the client's hostname, or 'localhost'. After accepting the control connection, subsequent connections (for ec_rpc and any peer queues) are checked to ensure that they are from the same client. If not, the attachment is terminated by ECL^{i}PS^e. Using 'localhost' as the name on either side will restrict the two sides to be on the same machine (and they must both use localhost for Host in the address, rather than the actual hostname).

10.3.2 An example

The following is a simple example of making a remote attachment on the remote side. In this case, the remote side is also an ECL^{i}PS^e program. A remote attachment by a program written in another programming language will need to provide a similar function in that language:

```
% Example code for making a remote attachment on the remote side

:- local variable(ecsidehost).

% remote_attach/6 makes a remote attachment to a host ECLiPSe program.
% Args:
%    +Host: host name on ECLiPSe side
%    +Port: port on ECLiPSe side
%    +Pass: 'pass-term' - used to verify the connection
%    +Init: goal to call to perform any application specific initialisation
%    -Control: the remote side (local) name of the control connection
%    -Ec_rpc:  the remote side name of the ec_rpc connection
%    -EcSideControl: the ECLiPSe side name for the control connection

remote_attach(Host, Port, Pass, Init, Control, Ec_rpc, EcSideControl) :-
     % control connection
     new_client_socket(Host, Port, Control),
     % send the protocol version information to the ECLiPSe side
     % we are the remote side, so must have our own version info.
     write_exdr(Control, remote_protocol(1)), flush(Control),
     % read response from ECLiPSe side to make sure it is compatible...
     read_exdr(Control, IsSameVersion),
     (IsSameVersion == "yes" ->
         true
     ;
         writeln("Incompatible versions of remote protocol. Failing...."),
```

```
         fail
     ),
     % send pass-term; if this does not match the pass-term on the ECLiPSe
     % side, it will terminate the attachment
     write_exdr(Control, Pass), flush(Control),
     % ECLiPSe side name for connection
     read_exdr(Control, EcSideControl),
     % send language name to ECLiPSe side
     write_exdr(Control, "eclipse"), flush(Control),
     % Ec_rpc connection
     new_client_socket(Host, Port, Ec_rpc),
     % read control name again on Ec_rpc to verify connection
     % in a more sophisticated implementation, this should time-out
     (read_exdr(Ec_rpc, EcSideControl) ->
        true
     ;
% if not verified, terminate connection
        close(Control), close(Ec_rpc),
        fail
     ),
     % Remote side now has control, call Init to perform application
     % specific initialisations
     call(Init),
     % hand control over to ECLiPSe side...
     write_exdr(Control, resume), flush(Control).


new_client_socket(Host, Port, Socket) :-
     socket(internet, stream, Socket),
     connect(Socket, Host/Port).

%-----------------------------------------------------
% the following code make use of remote_attach/6 to make an attachment,
% and then disconnect immediately when ECLiPSe side returns control

test(Host, Port) :-
     % make the remote attachment. For this simple example, there is no
     % application specific initialisations, and the default pass-term
     % of an empty string is used...
     remote_attach(Host, Port, "", true, Control, Ec_rpc, _),
     % control has been given to ECLiPSe side,
     % wait for ECLiPSe side to return control....
     read_exdr(Control, _),
     % immediately disconnect from ECLiPSe side
     write_exdr(Control, disconnect), flush(Control),
     % wait for ECLiPSe side to acknowledge disconnect...
     read_exdr(Control, disconnect_yield),
```

```
% clean up
close(Control), close(Ec_rpc).
```

In addition to the attachment, the example contains a very simple example of using the protocol by exchanging control messages with the ECL^{i}PS^e side. After attachment, it disconnects the remote attachment as soon as control is handed back to it. For more details on the messages, see next section.
To try out the example, start an ECL^{i}PS^e session and initiate a remote attachment. This process is the ECL^{i}PS^e side:

```
(ECLiPSe side)
[eclipse 1]: remote_connect(Host/Port, Control, _).
Socket created at address chicken.icparc.ic.ac.uk/32436
```

Now start another ECL^{i}PS^e session, compile the example program, and make the attachment to the ECL^{i}PS^e side:

```
(Remote side)
[eclipse 4]: test('chicken.icparc.ic.ac.uk', 32436).
```

On the ECL^{i}PS^e side, `remote_connect/3` should now succeed:

```
(ECLiPSe side)
Host = 'chicken.icparc.ic.ac.uk'
Port = 32436
Control = peer3
yes.
[eclipse 2]:
```

The remote side is suspended, awaiting the ECL^{i}PS^e side to return control. To return control to the remote side, the ECL^{i}PS^e side should call **remote_yield/1** (see section 10.6 for a description of `remote_yield/1`):

```
(ECLiPSe side)
[eclipse 2]: remote_yield(peer3).
Abort
[eclipse 3]:
```

In this simple example, when control is returned to remote side, it immediately disconnects, thus the `remote_yield/1` on the ECL^{i}PS^e side is aborted (as disconnect was initiated from the remote side), as per the disconnect protocol.

```
(Remote side)
[eclipse 4]: test('chicken.icparc.ic.ac.uk', 32436).

yes.
[eclipse 5]:
```

10.4 Remote Peer Queues

As discussed in section 10.2, peer queues can be formed interactively during an attachment between the two sides. The protocol provides for the creation and closing of these queues on both sides.
The handling of data on these queues are performed by *data handlers*, routines which either provide data to the queue or consume data from the queue. They are triggered by the appropriate control messages, so that a data consumer handler would consume data arriving on a queue, and a data provider handler would send data onto a queue (which would be consumed on the other side). These data handlers can be user defined.
The programmer for the remote side needs to provide the remote side of the interface to the peer queue.

10.4.1 Synchronous peer queues

The implementation of a synchronous peer queue on the ECL^{i}PS^e side is shown in figure 10.2. There is an in-memory buffer, which is implemented as a memory queue in ECL^{i}PS^e (i.e.

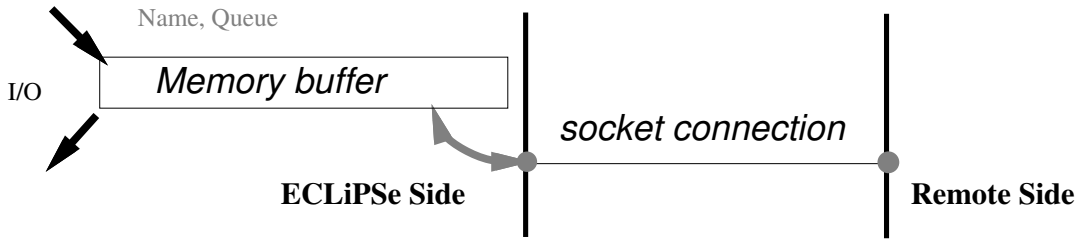


Figure 10.2: ECL^{i}PS^e implementation of remote synchronous peer queue

`queue("")` option for **open/3**). Associated with the memory queue is the actual socket stream to the remote side. The memory queue is the queue that the user sees on the ECL^{i}PS^e side, with name *Name* and a unique integer id *Queue*, which is the stream id for the memory queue. The id is used by both sides to identify the queue, as it is unique (a symbolic name can always be reassigned), and is used in the control messages. To conform to normal ECL^{i}PS^e streams, these queues appear uni-directional to the user, i.e. data can be either written to or read from the queue. The direction is *fromec* if the direction of the data is from ECL^{i}PS^e to the remote side (i.e. ECL^{i}PS^e can output to the queue), and *toec* if the direction is from the remote side to ECL^{i}PS^e (i.e. ECL^{i}PS^e can read from the queue). The user perform normal I/O operations on the memory queue, and the protocol ensures that the data is transferred correctly to the remote side without blocking.
The socket stream is largely hidden from the user on the ECL^{i}PS^e side, with data automatically transferred between it and the buffer. The buffer is needed to ensure that when data is transferred between the two sides, the I/O operation is synchronised (the data is being read on one end of the socket while it is being written at the other), so no blocking will occur. When a side needs to initiate an I/O operation with the other side (either to write data to or read data from the other side via the socket stream), it must have control (otherwise it would not be in a position to initiate an action). Before performing the I/O operation, a control message is sent to the other side to allow it to prepare to read the data before the I/O operation is actually initiated.

The memory queue thus serves to buffer the data before it is transferred to the socket stream. The socket stream must be connected to the remote side before it could be used. Control messages are used also to synchronise the connection of the socket. The protocol does not specify if a buffer is needed on the remote side, this depends on the facilities available in the language used for the remote side.
For a particular synchronous queue, a data handler can only be defined on one side of the queue, but not both.

10.4.2 Asynchronous peer queues

I/O operations on these queues do not need to be controlled by the remote protocol, but their creation and closing is controlled by the remote protocol, so that the remote interface can keep track of these queues. These are implemented as raw socket streams on the ECL^{i}PS^e side, with I/O operations performed directly on the socket. The Name and Queue id for the stream is thus those for the ECL^{i}PS^e socket stream. It is the responsibility of the user to ensure that I/O operations do not block on these queues. They are provided to allow asynchronous transfer of data (e.g. from ECL^{i}PS^e side to the remote side without handing over control), and also they allow more efficient transfer of data.

10.5 Control Messages

These are the messages that are exchanged between the ECL^{i}PS^e and remote sides when the remote side is attached. The messages are sent on the control connection, and are in EXDR format. All arguments for the messages are either atoms or integers. The messages are used to co-ordinate and synchronise the two sides. A message should only be sent from a particular side when that side has control, and control is handed over to the other side when the message is sent.
Most messages are used to initiate an *interaction* with the other side. That is, used to cause some action to take place on the other side: control is handed over, the action takes place on the other side, and eventually control is handed back when the action is completed. Control can also be explicitly handed over from one side to the other so that the other side can initiate interactions. Some additional messages can only be sent as part of an interaction, for exchanges of information between the two sides. Finally, there are messages for terminating the remote attachment. Note that interactions can be nested, that is, an interaction from one side can contain an interaction initiated from the other side.
The implementer for the remote interface should provide the methods for a programmer to initiate the interactions from the remote side. These routines would send the appropriate control messages to the ECL^{i}PS^e side, and the messages should not be directly visible to the programmer.
The usage of each message is summarised in the diagram(s) accompanying them. These diagrams show:

- time proceeds downward. The ECL^{i}PS^e side is shown on the left, the remote side on the right. Messages are shown as vertical arrows between the two sides. The direction of the arrow indicates the direction the message is sent.

- the context in which the message can be sent, i.e. the message from the other side that

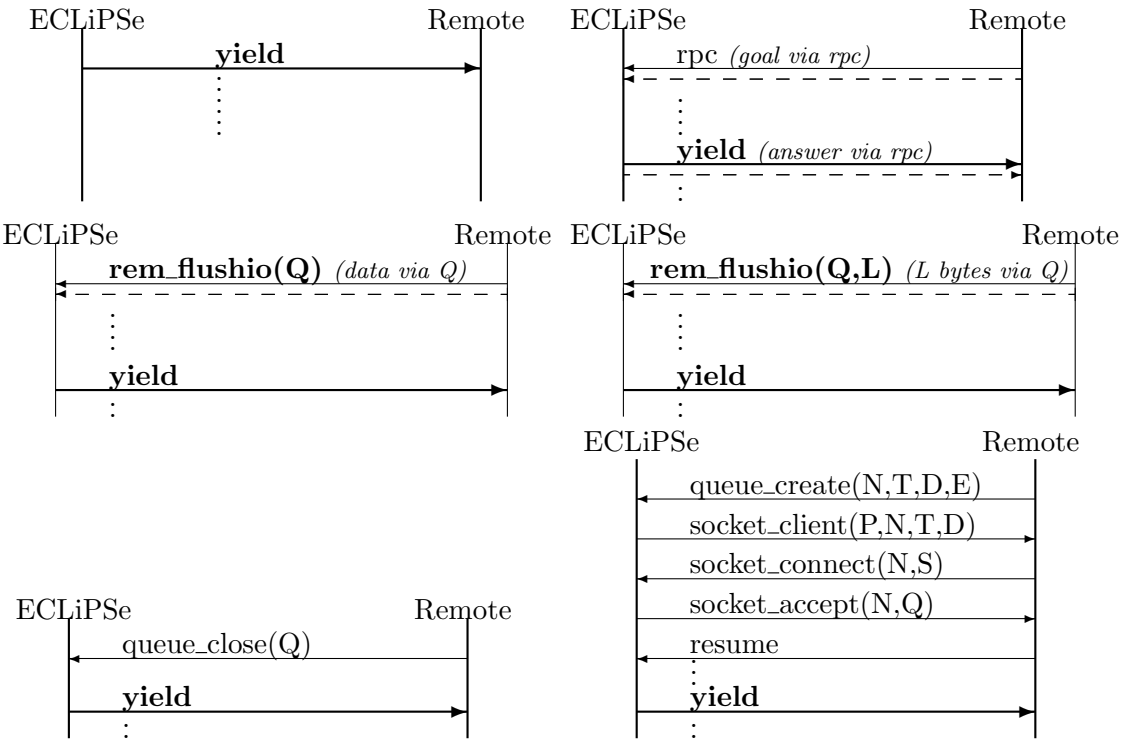
it is either expecting as a response, or that it is a response to. The message sequence is shown, with the message highlighted.

- any accompanying actions expected with the message. These actions are either sending or receiving data on some other connections between the two sides. These are shown as dashed arrows in the diagrams.

- whether nested interactions can take place between messages of an interaction. This is indicated by vertical ellipsis between the messages in the diagram. In such cases, the nested interaction can be initiated, and this interaction completed before the next message for the original interaction is expected.

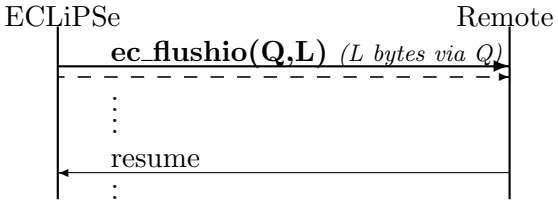
10.5.1 Messages from ECL^{i}PS^e side to remote side

yield this yields control to the remote side. The message is used either to implicitly return control to the remote side at the end of an interaction initiated from the remote side, or it is used to explicitly hand over control to the remote side.



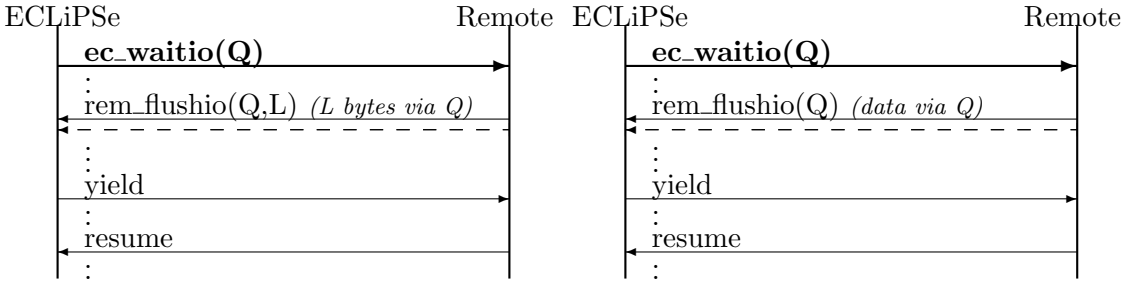
The interaction-initiating messages from the remote side will be described in more detail in their own sections.

ec_flushio(Queue, Length) this message is sent when output on a remote synchronous queue is flushed on the ECL^{i}PS^e side:



Queue is the ECL^{i}PS^e stream number for the peer queue, and Length is the number of bytes that is being sent on the queue. Control is yielded to the remote side. The data on the queue Queue will be sent through the queue after sending this message on the control connection, so on receiving this message on the remote side, the remote side should read Length bytes from Queue. After processing the data, the remote side should return control to the ECL^{i}PS^e side via a **resume** message.

ec_waitio(Queue) this message is sent when ECL^{i}PS^e requests input from a remote synchronous queue, and the data is not available in the queue's buffer.



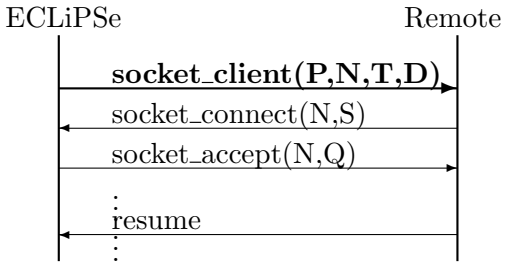
This interaction is triggered when the ECL^{i}PS^e side attempts a read operation on the empty buffer of a peer queue. The operation is suspended, and control is yielded to remote side so that it can provide the required data. There should be a data-provider handler associated with the queue on the remote side. This handler should obtain the data, and send the data to the ECL^{i}PS^e side. The data will arrive from the remote side via a **rem_flushio** message, which initiates a remote flushio interaction, nested within the ECL^{i}PS^e waitio interaction. The data arrives on the socket associated with the remote peer queue Queue, and is automatically copied by ECL^{i}PS^e into the peer queue buffer. Control is then yielded back to the remote side, completing the flushio interaction. The remote side then hands control back to ECL^{i}PS^e side by the resume message. The suspended read operation is resumed on the now non-empty buffer.

The remote flushio interaction is described in more detail in its own section. The main difference between a remote flushio initiated on the remote side and one initiated by an ECL^{i}PS^e waitio described here is that there must not be a data-consumer handler on the ECL^{i}PS^e side, as the data is to be consumed by the suspended read operation instead. This is ensured in the protocol by prohibiting handlers on both sides of a synchronous peer queue.

Note that the ECL^{i}PS^e side will also listen to the control connection while waiting for the data to be sent from the remote side. If a **resume** is sent before the data arrives, this is likely caused by a programming error in the data provider handler, which finished without sending data to the ECL^{i}PS^e side. The ECL^{i}PS^e side will print a warning message on the

warning output stream, and immediately yield back to the remote side. Other messages are handled as normal, recursively while waiting for the data to arrive – this is mainly intended to allow for unexpected aborts from the remote side, although it could also be used to perform ec_rpc calls before the remote side sends the data.

socket_client(Port, Name, Type, Dir) this requests the remote side to form a client socket connection for the remote peer queue Name. The queue is of type Type (sync or async), and direction Dir (fromec, toec for synchronous queues, bidirect for asynchronous queues). The client socket is to connect at port Port with the ECL^{i}PS^e side host name.



The ECL^{i}PS^e side first creates a server socket for the peer queue Name. The port address is Port. This, along with the details of the queue is passed to the remote side via the **socket_client** message. The remote side should then connect a client socket with Port as the port, and the Host used for the initial attachment (which is either localhost or the hostname of the `eclipse` side) for the host. It should also perform any additional setups for the peer queue using the information sent with the message (typically this involves setting up book-keeping information for the queue on the remote side). When the remote side connection is established, it returns control to ECL^{i}PS^e via a **socket_connect** message:

socket_connect(Name, Status)

Name is the name of the queue, and should be the same as the Name sent by the socket_client message. This is used to verify that the messages refer to same interaction. The ECL^{i}PS^e side will raise an error and disconnect from the remote side if the name does not match. Status is either success or fail, depending on if the remote side successfully created the remote side of the queue or not.

If Status is success, then the ECL^{i}PS^e side will complete the connection for the peer queue by accepting the socket connection. Since the remote end of the socket exists, the accept operation should succeed very quickly. If not, the operation will time-out, using the time-out interval specified when the attachment was made. The server socket is closed immediately after the accept operation. On successful connection, ECL^{i}PS^e first checks that this client's host is indeed the same as the one previously recorded for the remote side. If so, the ECL^{i}PS^e will finish creating the ECL^{i}PS^e side of the queue. If not, the connection is closed, and the operation is considered to have failed.

If Status from the socket_connect message is fail, then the ECL^{i}PS^e side will clean up the preparation for the peer queue.

The ECL^{i}PS^e side then returns control to the remote side via a **socket_accept** message:

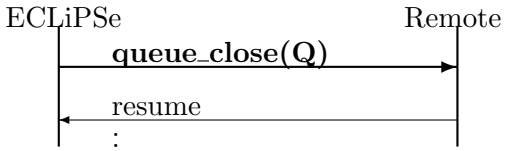
socket_accept(Name,Queue)

Name is again the name of the queue, and Queue the stream id. If the accept was unsuccessful (or if Status for socket_connect was fail), then Queue will be the atom fail, indicating that the peer queue connection was unsuccessful.

The remote side should then record the id Queue for later use (it is needed for the control messages connected with this peer queue). If instead fail was received, then the remote side should clean up the attempted queue connection. When the remote side has finished the final stage of the connection, control is returned to the ECL^{i}PS^e side via a **resume** message, and the socket_client interaction completes.

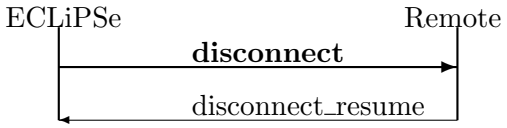
Note that the **socket_connect** and **socket_accept** messages are always exchanged during a socket_client interaction, even if the connection failed on the remote side before **socket_connect** is sent. They also can only occur in this context. If these messages occur in any other occasion, an error should be raised.

queue_close(Queue) this message is sent when the ECL^{i}PS^e side closes the peer queue with id Queue.



The remote side should close the remote side of the peer queue Queue, and remove all bookkeeping information associated with it. Control should then be returned to ECL^{i}PS^e via a **resume** message. The ECL^{i}PS^e side should also close the queue and remove bookkeeping information on the ECL^{i}PS^e side.

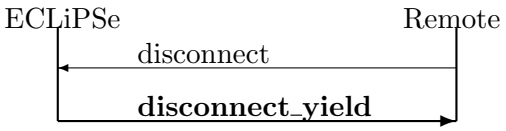
disconnect this message is sent when ECL^{i}PS^e side initiates disconnect.



Control is yielded to the remote side, which should acknowledge with the **disconnect_resume** message. Once the ECL^{i}PS^e side receives this message, the connection between the two sides is considered terminated. The ECL^{i}PS^e side will then close all the connections (the control and ec_rpc connections, and any asynchronous and synchronous queues) to the remote side, and clean up the information associated with the attachment. After sending the disconnect_resume message, the remote side should also shutdown its end of the connection by closing all the connections on its side.

Note that the disconnection message can be used to terminate the attachment from within an interaction. In such cases, the interaction(s) would not be completed.

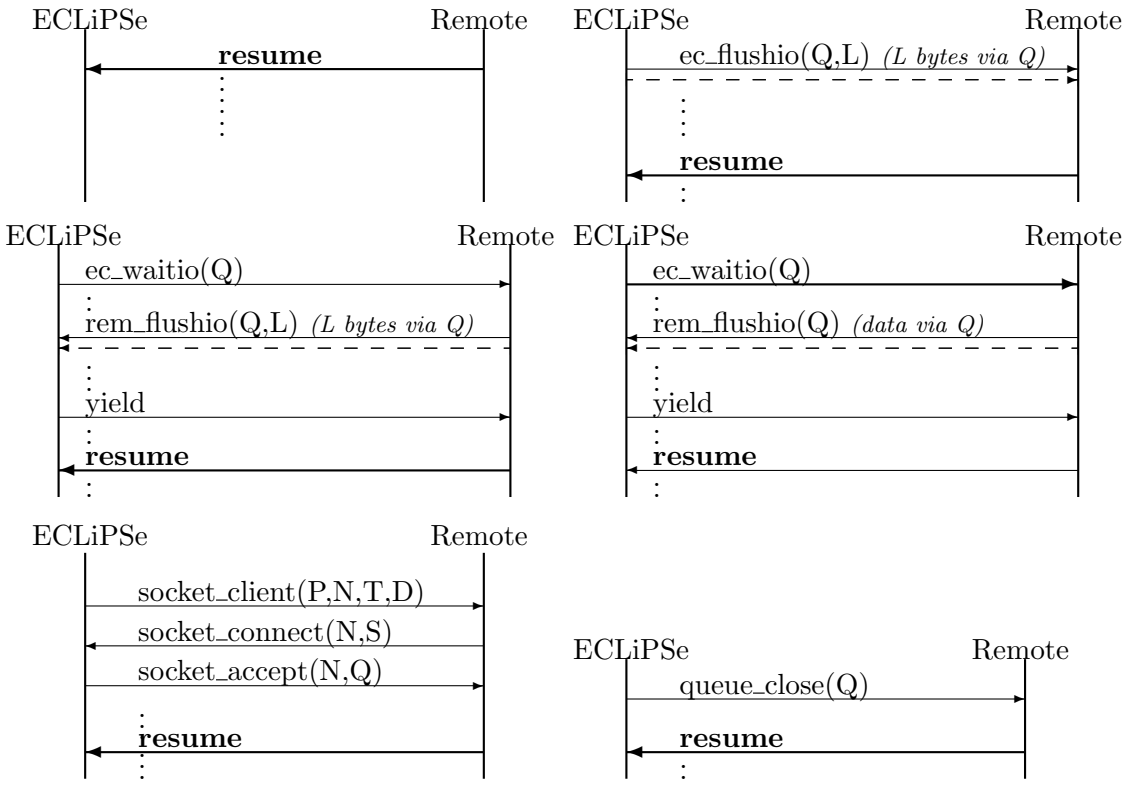
disconnect_yield this message is sent when ECL^{i}PS^e side receives a **disconnect** message from the remote side, i.e. the remote side initiated disconnection.



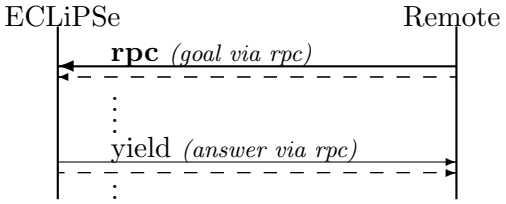
This message is sent as an acknowledgement to the disconnect message. Once the disconnect_yield is sent, the connection is considered terminated, and the ECL^{i}PS^e side will close all the connections and clean up. After the clean up, abort is called. This is done because the application on the ECL^{i}PS^e side (such as the remote development tools) may be deep inside some interaction loop with the remote side, and abort is the most general way of escaping from such a loop. It can be caught (see **remote_yield/1** in section 10.6) if the user wants a more graceful termination.

10.5.2 Messages from remote side to ECL^{i}PS^e side

resume this message hands over control from remote side to ECL^{i}PS^e side. This is used to either implicitly return control to the ECL^{i}PS^e side at the end of an interaction initiated from the ECL^{i}PS^e side, or to explicitly hand over control to the remote side.

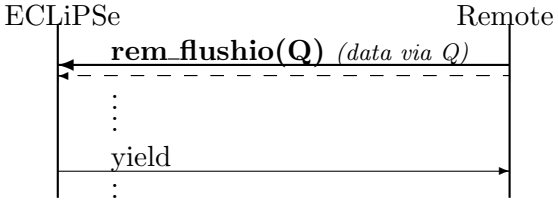


rpc this message is sent before the remote side sends an ec_rpc goal on the rpc connection.



After sending the **rpc** message, the remote side should then send the ec_rpc goal (in EXDR format) on the rpc connection. When the execution of the ec_rpc goal is finished, the ECL^{i}PS^e side will yield control back to the remote side with a **yield** message, followed by the result of the ec_rpc execution on the rpc connection (in EXDR format) – the goal with its bindings if the execution succeeded; 'fail' if the goal failed; 'throw' if an exception is generated.

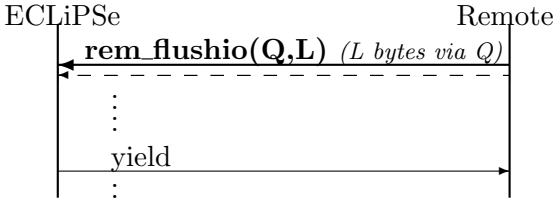
rem_flushio(Queue) this message is sent if the remote side wishes to transfer data to the ECL^{i}PS^e side on peer queue with queue id Queue, and the remote side does not know how many bytes will be sent with the operation.



After sending the message, control is transferred over to the ECL^{i}PS^e side, and the data is sent on the socket stream. On the ECL^{i}PS^e side, if the queue is a synchronous queue, then the data sent must be a single EXDR term, because otherwise the ECL^{i}PS^e side would not know when the data transfer is complete. The ECL^{i}PS^e side would read the data from the socket stream as a single EXDR term, which is then written onto the buffer. If an event handler has been associated with the peer queue, this will now be invoked to consume the data from the buffer. If not (for example, if the **rem_flushio** was initiated by an **ec_waitio** message), then the data is left on the buffer to be processed later.

The **rem_flushio** message can also be used to sending data to ECL^{i}PS^e for asynchronous queues as well. In this case, an event handler is directly associated with the socket stream, and this event is invoked when the rem_flushio message is received. The event handler goal in this case is invoked with the 'culprit' argument being the term `rem_flushio(Queue, Len)`, where Len is the atom 'unknown'. It is up to the user-defined event handler goal to properly read the data: since the length is unknown, the data sent should have natural boundaries, e.g. EXDR terms, or use a mutually agreed 'end of data' marker.

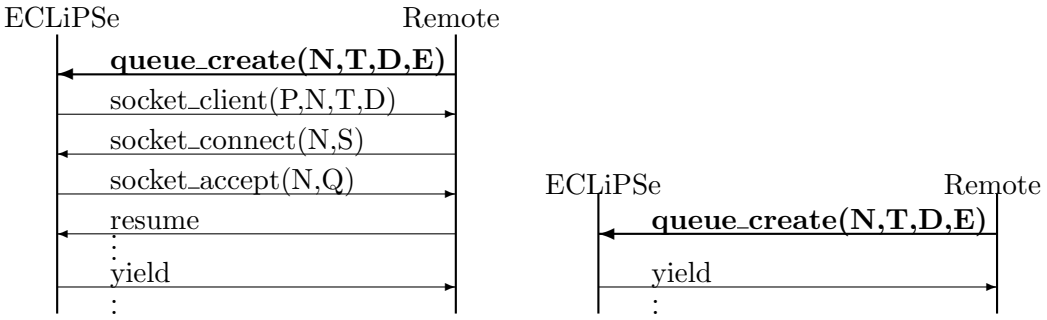
rem_flushio(Queue, Length) this message is sent if the remote side wishes to transfer data to the ECL^{i}PS^e side on peer queue with queue id Queue, and the length of data to be sent is known, and is specified in Length (the number of bytes to be sent).



After sending the message, control is transferred over to the ECL^{i}PS^e side, and the data is sent on the socket stream. On the ECL^{i}PS^e side, if the queue is a synchronous queue, then it would read Length bytes of data from the socket stream and transfer the data to the queue buffer. If an event handler has been associated with the peer queue, this will now be invoked to consume the data from the buffer. If not (for example, if the **rem_flushio** was initiated by an **ec_waitio** message), then the data is left on the buffer to be processed later.

In the case that the peer queue is an asynchronous queue, an event handler is directly associated with the socket stream, and this event is invoked when the rem_flushio message is received. The event handler goal in this case is invoked with the 'culprit' argument being the term `rem_flushio(Queue, Length)`, It is up to the user-defined event handler goal to properly read the data.

queue_create(Name, Type, Dir, Event) this message is sent when the remote side wishes to initiates the creation of a new peer queue. Name is the name of the peer queue, Type is its Type: sync for synchronous, async for asynchronous. Dir is the data direction: fromec or toec, and Event is the name of the event that will be raised for the event handler goal on the ECL^{i}PS^e side, if no event is to be associated with the queue, this should be the empty atom (`''`).

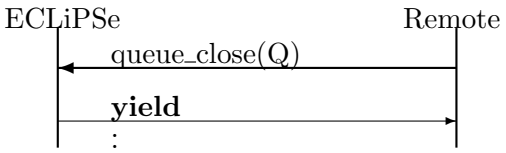


Control is handed over to ECL^{i}PS^e side, which should then set up a new server socket for connecting the socket stream for the peer queue. Once this server socket is set up, the creation of the queue proceeds via a socket_client interaction from the ECL^{i}PS^e, i.e. the ECL^{i}PS^e side sends a **socket_client** message. For more detail, see the description for the socket_client message. At the end of the socket_client interaction, the peer queue would be established, and ECL^{i}PS^e side has control. The ECL^{i}PS^e side will yield control back to the remote side, completing the queue_create interaction.

Note that the **socket_client** interaction is performed by the ECL^{i}PS^e built-in **peer_queue_create/5**, this is the goal that ECL^{i}PS^e calls on receiving the **queue_create** message.

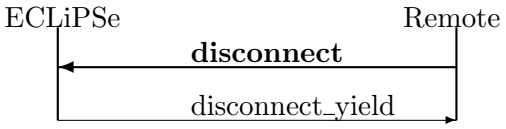
If the initial creation of the socket server fails, then the ECL^{i}PS^e side will not initiate a socket_client interaction. Instead, it will simple yield control back to the remote side with a **yield** message. In this case, no peer queue is created.

queue_close(Queue) this message is sent when the remote side closes the peer queue with id Queue.



The ECL^{i}PS^e side should close the ECL^{i}PS^e side of the peer queue Queue, and remove all bookkeeping information associated with it. Control should then be returned to ECL^{i}PS^e via a **yield** message. The queue should also be closed on the remote side, with the bookkeeping information removed too.

disconnect this message is sent if the remote side wishes to initiate disconnection.



Control is handed over to the ECL^{i}PS^e side, which will acknowledge with **disconnect_yield** message. Once the ECL^{i}PS^e side receives this message, the remote attachment between the two sides is considered terminated. The remote side should now close all connections to the ECL^{i}PS^e side. Concurrently, the ECL^{i}PS^e side will also close down its end of the connections.

The **disconnect** message can be issued during an interaction. In such cases, the interaction will be terminated early along with the attachment.

disconnect_resume this message is sent in acknowledgement of a disconnection initiated from the ECL^{i}PS^e side.

After sending this message, the remote attachment between the two sides is considered terminated. The remote side should now close all connections to the ECL^{i}PS^e side. Concurrently, the ECL^{i}PS^e side will also close down its end of the connections. In addition, this message should be sent if the remote side has to terminate the attachment while the ECL^{i}PS^e side has control. This can happen if the remote process is forced to quit. This is the only case where a message can be sent via the control connection on the remote side while it does not have control. Once the message is sent, the remote side can terminate its connection unilaterally.

10.5.3 The disconnection protocol

Under normal circumstances, the disconnection of the two sides is initiated by the side that has control, by sending a **disconnect** message to the other side. The other side acknowledges this by responding with a **disconnect_yield** (ECL^{i}PS^e side) or **disconnect_resume** (remote side). The acknowledgement should be sent when that side is ready to disconnect. Once the messages have been exchanged, both sides should be ready, and can physically disconnect. The exchange of messages should ensure that any asynchronous I/O between the two sides are properly terminated.
However, under some circumstances, a side may be forced to disconnect when it does not have control. For example, in the Tcl remote interface, the root window for the Tcl process may be destroyed by the user. In such cases, a unilateral disconnect will be performed by that side – only the second part of the normal disconnect protocol is performed by sending the disconnect acknowledge message (**disconnect_resume** or **disconnect_yield**) without being initiated by a **disconnect** message.
The ECL^{i}PS^e side checks the control connection for any unexpected incoming messages before it sends an outgoing control message. If there is a **disconnect_resume** message, the ECL^{i}PS^e side will perform the disconnection on its side.
When the user exits normally from an ECL^{i}PS^e session, ECL^{i}PS^e will disconnect from all remote attachments. This is done in **sepia_end/0** event handler.
As part of the disconnection process on the ECL^{i}PS^e side, a user definable event will be raised in ECL^{i}PS^e, just before the remote queues are closed. This allows the user to define application specific handlers for dealing with the disconnection of the remote interface (on the ECL^{i}PS^e side). A similar handler should probably be provided on the remote side. The event raised has the same name as the control stream. The event handler for this event is initially defined to be `true/0` (i.e. a no-op) when the remote connection is set up. The handler can then be redefined by the user, e.g. during the user-defined initialisation during attachment:

```
...
remote_connect(localhost/MyPort, Control,
    set_event_handler(Control, my_disconnect_handler/1)),
...

my_disconnect_handler(Remote) :-
% just print out a message about the disconnection
    printf("Disconnected from remote attachment %w", [Remote]).
```

10.6 Support for the Remote Interface

ECL^{i}PS^e provides the following predicates to support the remote interface:

remote_connect(?Address, ?Peer, ?InitGoal) Initiates a remote attachment at address Host/Port. The predicate will wait for a remote process to establish an attachment according to the protocol described in section 10.3. If instantiated, InitGoal is called after the connection is established to perform any user defined initialisation on the ECL^{i}PS^e side (this can be used for example to define a disconnection handler that will be called when

the two sides disconnect). The predicate succeeds when the attachment is successfully made and the remote side returns control to the ECL^{i}PS^e side. Peer is the name of the control connection, and is used to identify a particular remote peer (an ECL^{i}PS^e session can have several).

remote_connect_setup(?Address, ?Peer, -Socket) `remote_connect/2` is implemented by calls to `remote_connect_setup/3` and `remote_connect_accept/6`. These lower level predicates allow more flexibility in the implementation of the remote attachment, at the cost of some increased complexity.

The two predicates must be used together, with `remote_connect_setup/3` called first. The predicate creates a socket server for remote attachment at host Host and port Port. `Socket` will be instantiated to the name of the socket server that is created. When the predicate returns, the remote process can request the socket connection at Host/Port address (if the request is issued before `remote_connect_setup/3` is called, the server would refuse the connection). The remote process will suspend waiting for the request to be accepted. This will happen when `remote_connect_accept/6` is called.

Splitting the attachment into two predicates enables the user to start the remote program in between. This will allow the user to start the remote attachment automatically by executing the remote program from within ECL^{i}PS^e with an **exec/3** call.

remote_connect_accept(?Peer, +Socket, +TimeOut,?InitGoal,?PassTerm,-InitRes) This predicate accepts an remote attachment at the socket server Socket. This predicate is called after a call to `remote_connect_setup/3`, with the same arguments for Peer and Socket. The predicate will create the control and rpc connections according to the protocol described in section 10.3 with a remote process. `Socket` will be closed if the attachment is successful.

`TimeOut` specifies the amount of time (in seconds) that the predicate will wait for a remote process to attempt a remote attachment. If no remote attachment request is made in the specified time, the predicate will fail. This time is also used for the time-outs on peer queue connections. To make the predicate block indefinitely waiting for a remote attachment, the atom `block` can be used for `TimeOut`.

`InitGoal` is used to define the optional initialisation on the ECL^{i}PS^e side when the two sides are connected. InitGoal will be called immediately after connection, before the two sides are allowed to interact. If no initialisation is desired, then the argument can be left uninstantiated. The result of executing the goal is returned in `InitRes`, which should be initially left uninstantiated.

`PassTerm` is the pass-term that is used to verify the connection. The remote side sends a pass-term which is checked to see it is identical to `PassTerm`. If not, the attachment fails.

Unimplemented functionality error is raised if the remote protocol used by the remote and ECL^{i}PS^e sides are incompatible. This can happen if one side is outdated (and using an outdated version of the protocol).

remote_disconnect(+Peer) Initiates the disconnection of the remote attachment represented by Peer. All connections (control, ec_rpc, synchronous and asynchronous streams) will be closed. The predicate succeeds after the clean up. In addition, a `Control` event will be raised.

remote_yield(+Peer) Explicitly yield control to the remote side Peer. Execution on ECL^{i}PS^e side will be suspended until the remote side returns control to ECL^{i}PS^e side. The predicate will succeed when remote side returns control. The predicate will abort if the remote side initiates disconnect. The abort will occur after the remote attachment is disconnected. The abort can be caught to allow for more graceful exits in user applications by wrapping the `remote_yield/1` in a **block/3** call.

peer_queue_create(+Name,+Peer,+Type,+Direction,+Event) Creates a peer queue from ECL^{i}PS^e. `Name` is the name for the queue, and `Peer` is the peer to which the queue would be connected. `Type` specifies if the queue is synchronous or not (atom sync or async), and direction is the direction of the queue (fromec, toec for synchronous queues, it is ignored for asynchronous queues), `Event` is the name of the event that will be raised on the ECL^{i}PS^e side. The user should associate an event handler goal with Event. If no event is to be raised, then the empty atom (`''`) should be used.

The predicate does not provide a way to specify a handler for the queue on the peer side. This is because it is not possible to provide a generic way that is independent of peer's programming language.

peer_queue_close(+Name) Closes the peer queue Name from ECL^{i}PS^e.

Chapter 11

DBI: ECL^{i}PS^e SQL Database Interface

11.1 Introduction

The ECL^{i}PS^e SQL database interface is a low-level interface to the SQL language. As far as possible it has been attempted to give the full power of the SQL interface to the programmer. A number of features are designed to permit transfer of large amounts of data with minimal overhead:

- The buffered retrieval of multiple tuples from a relation or view.
- Buffered multiple updates and inserts
- Re-use of SQL statements that are parametrised with placeholders.
- Tuple templates to permit the allocation and re-use of fixed size buffers.

The interface provides safe handles to database cursors. Higher level interfaces can be constructed on top of this. For example:

- Viewing an SQL query as a predicate that yields different tuples on backtracking.
- Viewing an SQL query as a lazily constructed list of tuples.

On backtracking, cursors as well as database sessions are automatically closed. This is required if one is to build abstractions where they are hidden from the programmer.
Nowhere in the interface are SQL commands ever looked into. They are always passed straight from the user to the database. This means that any SQL command supported by the underlying database will be accessible. Note also any differences in the SQL between different databases are also not hidden from the user, so the SQL written by the user must be valid SQL for the database that is being interfaced to.
When retrieving or inserting tuples, tuple templates are used to specify the types of the different fields being retrieved or inserted. This allows for a flexible mapping between database internal types and the tuples retrieved. For example one can retrieve a date as either a string or an atom. What type combinations are supported will depend on the underlying database.

It is possible to write SQL queries with parameters (SQL placeholders), so that one can use the same query several times, with different values. This feature, together with the availability of any number of cursors can be used to write complex queries not expressible with a singe SQL statement.
If the database has a generic binary format, such as Binary Long Object (BLOB) fields, one can store and retrieve arbitrary terms in them automatically using the interface. To exchange ECL^{i}PS^e terms with other applications via the database, the terms can be converted to EXDR data interchange format (see the Embedding and Interfacing chapter for details) before storing them in the database.
It is possible to open several sessions to different databases simultaneously. Transactions apply to one session only though.
The code is written so that it will be relatively easy to extend it to interface to different kinds of databases. The currently supported database is MySQL.

11.2 Using the SQL database interface

The SQL database interface is contained in the dbi library.

```
[eclipse 1]: lib(dbi).
...
yes.
```

Your environment must be set up so that you can connect to a database supported by lib(dbi). Normally a database administrator will have written a script to do this automatically.
MySQL specific note: on some platforms, the MySQL client library is provided as a dynamic load library (libmysqlclient.so in most Unix systems, libmysql.dll in Windows systems). This file is not provided with ECL^{i}PS^e distribution as it is part of the MySQL system, and is covered by its own license. MySQL can be downloaded from `http://dev.mysql.com/downloads`. To successfully load the dbi library in such cases, you must have a version of this library file that match the one that your copy of ECL^{i}PS^e was compiled with, and in addition, ECL^{i}PS^e must be able to find this library file when the dbi library is loaded. Normally, the person who installed your ECL^{i}PS^e system should also make sure that ECL^{i}PS^e can find this client library file using the normal way that the operating system can find dynamic load libraries, e.g. by putting the file in a standard system library location, or in the ECL^{i}PS^e dynamic library directory for the platform in your ECL^{i}PS^e directory. On Unix systems, it is also possible for the user to set an environment variable (usually `LD_LIBRARY_PATH`) to point to where the dynamic library is.

11.3 Data Templates

If supported by the database, the interface allows the use of prepared SQL statements with parameters (placeholders). Prepared SQL statements are pre-parsed by the database, and can be executed more efficiently for multiple times, with the placeholders acting like variables, taking on different values for each execution.
The syntax used for prepared statements is that provided by the database, but a common syntax is to use `?` to indicate a placeholder. For example:

```
insert into employees (enum, ename, esalary, ejob) values (?, ?, ?, ?)
```

would be used to add rows to the employees relation.
Such an SQL statement has to be prepared before execution. It can then be executed in batches to insert several tuples in one go. Preparation involves parsing the SQL statement and setting up a buffer for the tuples.
A data template is used as an example buffer. For the insert command above it might look like:

```
emp(1234,"some name",1000.0,'some job')
```

The argument positions correspond to the order of the placeholder in the SQL statement. The types of the data will be used to type-check the tuples when they are inserted.
The following ECL^{i}PS^e goal uses a template to create a cursor for an insert command:

```
SQL = "insert into employees (enum,ename,esalary,ejob) values (?,?,?,?)",
Template = emp(1234,"some name",1000.0,'some job'),
session_sql_prepare(H, Template, SQL, Cursor),
```

`H` is a handle to a database session, and `Cursor` is the cursor created for the prepared statement `SQL`.
The cursor can then be used to insert several rows into the employee table.

```
cursor_next_execute(Cursor,emp(1001,"E.G. SMITH",1499.08,editor)),
cursor_next_execute(Cursor,emp(1002,"D.E. JONES",1499.08,journalist)),
```

Similarly for queries a data template specifies the types of the columns retrieved. The positions of the arguments correspond to the position of the select list items. The example template above might be used for a query like

```
SQL = "select enum, ename, esalary, ejob from employees where esalary > 1000",
Template = emp(1234,"some name",1000.0,'some job'),
session_sql_query(H, Template, SQL, Cursor),
cursor_next_tuple(Cursor,Tuple),
% Tuple is now somthing like emp(1001,"E.G. SMITH",1499.08,editor)
```

If a structure or list appears in one of the argument positions this stands for a general term, to be stored or retrieved in external database format. This way one is not limited to atomic types which have natural mappings to database types.

11.3.1 Conversion between ECL^{i}PS^e and database types

Data is passed from ECL^{i}PS^e into the database via placeholders in prepared SQL statements, and passed from the database to ECL^{i}PS^e via the results tuples returned by executing SQL queries. The interface takes care of the conversion of data to/from ECL^{i}PS^e types to the external C API types, and the database then converts these to/from the internal types of the database, as determined by the SQL statement used. The exact internal database types, and the exact conversion rules between the C type and the database type is dependent on the database's API, but in general the following should hold for ECL^{i}PS^e types:

Strings and atoms are converted to C char * type. This should be used for non-numeric data. Restrictions may apply depending on the SQL datatype – for example, non-binary string types (such as VARCHAR) does not accept generic binary strings, and SQL data and time types must be in the correct syntax – consult the database manual for the syntax for these types.

Integers and Floats are converted to C integers and doubles, which are then converted to the specified SQL numeric types. The numbers are passed to the database's C API at the maximum precision and size supported by the database's API. Any integers outside the range representable by the C API's integer type will raise an error. Note that while the number passed to the database is at maximum precisiion and size, the corresponding SQL numberic type specified by the SQL statement that receives the number may be smaller (e.g. SMALLINT). The exact behaviour in this case depends on the database.

General terms are converted to ECL^{i}PS^e's internal dbformat - a flat binary representation of the term, and then to an appropriate SQL binary type. This allows ECL^{i}PS^e to store and retrieve general terms, but if it is required to exchange Prolog terms with external sources via the database, then the term should be first converted to EXDR format, and the EXDR string can then be passed to the database. Note that EXDR can only represent a subset of terms – see the Embedding and Interfacing manual for details.

11.3.2 Specifying buffer sizes in templates

Prolog terms, strings and atoms can have variable sizes, but when they are passed into and out of the database, they pass through fixed size buffers for reasons of efficiency.
In the case of fetching data from fixed size database fields, the size of these buffers is by default, the size of the field in the database. In the case of variable sized fields and of placeholders, a default size is chosen.
Rather than taking the default it is possible to write templates that specify a size. This is done by mentioning the size in the argument of the template.

'123' defines an atom datatype where the maximum length in bytes is 123. The length is given as a decimal number.

"123" defines a string datatype where the maximum length in bytes is 123. The length is given as a decimal number.

s(123,X) Describes any Prolog term that occupies up to 123 bytes in external database format. Any structure whose first argument is an integer can be used.

For example the following two templates specify the same type of tuple but the second one defines some sizes for the different elements in the tuple as well.

```
emp(1234,"name", Rules, job)
emp(1234,"10",rules(4000),'10')
```

The size information is used to define the minimum size of the buffer used to pass data between ECLiPSe and the database. Depending on the database and the situation (input/output, prepared/direct statements), such an intermediate buffer may not be used; in such cases, the buffer size will be ignored. Otherwise, if the data is too big to fit into the buffer, an error will be raised. The data in the buffer is passed to/from the database, which may have its own size specification for the data, which is independent of the size information specified in the template. In the case of sending data to the database, and the data is too large for the database, the exact behaviour is dependent on the database. In the case of receiving the data from the database, and the data is too large for the buffer, an error will be raised.

11.4 Built-Ins

11.4.1 Sessions

These predicates deal with sessions as a whole. A session is used to identify a connection to a database. Associated to that session are a number of cursors. These are handles to SQL statements which are currently being executed. For example a query where only some of the matching rows have been fetched.
Database transactions – where updates to the database are local to the session until committed, and where uncommitted changes can be rolled back, are supported if the external database supports transactions[1].

session_start(+Login, +Password, +Options, -Session)

This create a new session by establishing a new connections to the database server, and associates it with a handle, returned in Session.
The session remains in existence in subsequent goals. It is automatically closed if the program fails or aborts back beyond this call. The session is used as a access handle to the database in subsequent calls to this interface.
The automatic closure is particularly useful in case of a program aborting due to a runtime error. Closing the database ensures any database updates that have not been committed will be undone.

session_close(+Session)

This closes a session, disconnecting from the database server. It takes effect immediately. This allow resources allocated for the session to be freed. To free all resources associated with a session, all cursors of the session should also be closed with **cursor_close/1**.

session_commit(+Session)

If executed outside the scope of a **session_transaction/2** goal, this commits any transactional updates to the database made within Session. Otherwise, it simply succeeds without doing anything.

session_rollback(+Session)

If executed outside the scope of a **session_transaction/2** goal, this undoes all transactional changes made to the database since the last commit for this session. Otherwise, it will simply abort the complete outer transaction. (Note: not all changes can be rolled back; consult the DB manual for details)

session_transaction(+Session, +Goal)

This executes Goal as a database transaction. This predicate is only useful if the database supports transactions. Data base updates within Goal are committed if Goal succeeds; if Goal fails or aborts, the updates are rolled back.

[1]Some databases supports both transactional and non-transactional updates, and not all updates can be rolled back. Consult the database manual for more details

Calls of this predicate can be nested, but only the outermost transaction is real. All the inner transactions are simply equivalent to call(Goal). This way it is possible to write a call to session_transaction/2, into some code that implements a simple update, but then to include that simple update into a larger transaction.
Transactions are local to one session so there is no way to safely make an update relating to several sessions.

```
recorded_transfer(Session,Date,Amount,FromAccount,ToAccount) :-
    session_transaction(Session, (
        transfer(Session, Amount,FromAccount,ToAccount),
        check_overdraft_limit(FromAccount),
        record_transfer(Date,Amount,FromAccount,ToAccount)
    )).

transfer(Session, Amount,FromAccount,ToAccount) :-
     session_transaction(Session,
         transfer_(Session,Amount,FromAccount,ToAccount)
     ).
```

In the above example we can see two nested transactions. One simple bank transfer that is not recorded, and an outer transaction recording the occurrence of the transfer and checking the balance.
Since a nested transaction is simply a call of its goal, with no partial rollbacks care has to be taken not to redo transactions on failure unless one is sure one is at an outer transaction.

11.4.2 Database Updates

For database updates, lib(dbi) provides predicates to execute SQL statements on the database without returning results. **session_sql/3** executes an SQL statement directly. **session_sql_prepare/4** is used to prepare SQL statements, returning a cursor to the prepared statement, which can then be executed multiple times with different placeholder values using either **cursor_next_execute/2** or **cursor_all_execute/2** or **cursor_N_execute/4**. Cursors are automatically closed if the program backtracks or aborts beyond the predicate that created it. Alternatively, the cursor can be closed explicitly by **cursor_close/1**.
The datatypes of the parameters for the prepared statement is specified by a template given to session_sql_prepare/4. See section 11.3 for details on the templates.

session_sql(+Session, +SQL, -RowProcessedCount)

This is the simplest interface to execute an SQL statement with no placeholders.

```
make_accounts(Session) :-
    session_sql(Session,
        "create table accounts \
         (id           decimal(4)      not null,\
          balance      decimal(9,2)    default 0.0 not null, \
          overdraft    decimal(9,2)    default 0.0 not null \
```

```
        )" ,_),
    session_sql(Session,
        "insert into accounts (id,balance) values (1001,1200.0)",1),
    session_sql(Session,
        "insert into accounts (id,balance) values (1002,4300.0)",1).
```

In the example we see session_sql/3 used, first to create a table, and then to initialise it with two rows. The rows processed counts are checked to make sure exactly one row is processed per statement.
This code assumes a session with handle `Handle` has been started beforehand.

session_sql_prepare(+Session, +Template, +SQL, -Cursor)

This creates Cursor, which is a handle to the prepared statement, possibly with placeholders. Template specifies the types of the placeholders (see section 11.3).

```
transfer_(Session, Amount, FromAccount, ToAccount) :-
    SQL = "update accounts set balance = balance + ? \
                                            where id = ?",
    Deduct is - Amount,
    session_sql_prepare(Session,incbal(1.0,12),SQL,Update),
    cursor_next_execute(Update,incbal(Deduct,FromAccount)),
    cursor_next_execute(Update,incbal(Amount,ToAccount)).
```

In the example a cursor is prepared to modify account balances. It is used twice, once to deduct an amount and once to add that same amount to another account. Note: the example uses MySQL's syntax for prepared statement, which may differ from other databases. Consult your database manual for prepared statement syntax.

cursor_next_execute(+Cursor, +Tuple)

Execute the prepared SQL statement represented by Cursor, with Tuple supplying the values for any parameter values. This call can be executed any number of times on the same cursor.

cursor_all_execute(+Cursor, +TupleList)

The SQL statement of Cursor is executed once for each Tuple in TupleList. This could be defined as:

```
cursor_all_execute( Cursor, []).
cursor_all_execute( Cursor, [Tuple | Tuples] ) :-
    cursor_next_execute(Cursor, Tuple),
    cursor_all_execute( Cursor, Tuples ).
```

cursor_N_execute(+Cursor, -Executed, +TupleList, -RestTupleList)

Some databases supports the execution of multiple tuples of parameter values at once, doing this more efficiently than executing each tuple of parameter values one by one. This predicate is provided to support this.

Note that for databases that does not support execution of multiple tuples, this predicate is implemented by executing the Tuples one by one as in **cursor_next_execute/2**, and there is no gain in efficiency over using cursor_next_execute/2.

```
transfer_(Session, Amount, FromAccount, ToAccount) :-
    SQL = "update accounts set balance = balance + ? \
                                            where id = ?",
    Deduct is - Amount,
    session_sql_prepare(Session,incbal(1.0,12),SQL,Update),
    Updates = [incbal(Deduct,FromAccount),incbal(Amount,ToAccount)],
    cursor_N_execute(Update,_,Updates,[]).
```

The example shows how to re-code the bank transfer predicate from **cursor_next_execute/2**, to execute both updates with one call. This could lead to some performance improvement in a client server setting for databases that supports multiple parameter tuples.

11.4.3 Database Queries

For database queries, lib(dbi) provides predicates to execute SQL statements and extract the results returned by the database. **session_sql_query/4** or **session_sql_query/5** executes an SQL statement, returning a cursor to allow the results to be extracted from the database. The predicates to do this are **cursor_next_tuple/2**, **cursor_all_tuples/2** and **cursor_N_tuples/4**. The datatypes of the results tuple is specified by a template given to session_sql_query/4,5. See section 11.3 for details on the templates.

session_sql_query(+Session, +Template, +SQL, -Cursor)

This executes SQL and creates the handle Cursor for the SQL query. Template specifies the datatypes of the results tuples.

cursor_next_tuple(+Cursor, -Tuple)

A single tuple is retrieved from the database. Calling this predicate again with the same cursor will retrieve further tuples Any NULL values are returned as uninstantiated variables.
Once all the tuples have been retrieved this predicate fails.
If Tuple does not unify with the retrieved tuple, the predicate fails.

```
check_overdraft_limit(Session, Account) :-
    L = ["select count(id) from accounts \
        where     id = ",Account," and balance < overdraft"],
    concat_string(L,SQL),
    session_sql_query(Session,c(0),SQL,OverdraftCheck),
    cursor_next_tuple(OverdraftCheck,c(Count)),
    Count = 0.
```

In this example a query is built to verify that the balance of an account is not less than its overdraft facility. All comparisons are done within the database, and we are just interested in checking that no rows match the 'where' clause.
For this kind of application one would not normally use concat_string/2. SQL placeholders would be used instead. See **session_sql_prepare_query/5**.

cursor_all_tuples(+Cursor, -TupleList)

The SQL query represented by the cursor is executed and all the matching tuples are collected in TupleList.
This could be defined as:

```
cursor_all_tuples( Cursor, Tuples ) :-
    ( cursor_next_tuple(Cursor, T) ->
        Tuples = [T | Ts],
        cursor_all_tuples(Cursor, Ts)
    ;
        Tuples = []
    ).
```

cursor_N_tuples(+Cursor, -Retrieved, -TupleList, -RestTupleList)

If the underlying DB supports the retrieving mutule tuples in one go, then a buffer full of tuples matching the query is retrieved, otherwise all the remaining tuples are retrieved.
TupleList and RestTupleList form a difference list containing these tuples. Retrieved is the number of tuples retrieved.

11.4.4 Parametrised Database Queries

The library allow SQL queries to be prepared and parameterised, if prepared SQL statements are supported by the underlying database. Templates are needed for specifying the datatypes of the parameters (as with **session_sql_prepare/4**), and for the results tuples (as with **session_sql_query/4**. An SQL query is prepared by **session_sql_prepare_query/5**, it then needs to be executed by **cursor_next_execute/2** or **cursor_next_execute/3** (cursor_next_execute/3 allows the specification of options for the cursor), and the results can then be retrieved by **cursor_next_tuple/2**, **cursor_all_tuples/2** and **cursor_N_tuples/4**. After executing cursor_next_execute/2, it can be executed again with a new tuple of parameter values. Any unretrieved results from the previous execute are discarded. Note that this is non-logical: the discarded results are not recovered on backtracking.

session_sql_prepare_query(+Session, +ParamT, +QueryT, +SQL,-Cursor)

This creates Cursor, which is a handle to the prepared SQL query.
By changing the placeholders one gets a fresh query and can start extracting tuples again.
In this example a generic query is built to check whether an account is overdrawn, and a cursor for this query is created.

```
make_check_overdraft_limit(Session, Cursor) :-
    SQL = "select count(id) from accounts where ID = ? \
               and balance < overdraft",
    session_sql_prepare_query(Session,a(0),c(0),SQL,Cursor).

check_overdraft_limit(Check,Account) :-
    cursor_next_execute(Check,a(Account)),
    cursor_next_tuple(Check,c(Count)),
    Count = 0.
```

The check_overdraft_limit/2 predicate uses the cursor to check an account. This cursor can be re-used to check any number of accounts.

Appendix A

Parameters for Initialising an ECL^{i}PS^e engine

It is possible to parametrise the initialisation of the ECL^{i}PS^{e}engine by calling the functions ec_set_option_long() and ec_set_option_ptr(). This must be done before initialisation.

Installation directory

```
ec_set_option_ptr(EC_OPTION_ECLIPSEDIR, "/usr/tom/eclipse");
```

This can be used to tell an embedded ECL^{i}PS^e where to find it support files. The default setting is NULL, which means that the location is taken from the registry entry or from the ECLIPSEDIR environment variable.

Stack Memory Allocation

```
ec_set_option_long(EC_OPTION_LOCALSIZE, 128*1024*1024);
ec_set_option_long(EC_OPTION_GLOBALSIZE, 128*1024*1024);
```

The sizes in bytes of the stacks can be varied. They will be rounded to system specific pagesizes. On machines where initially only virtual memory is reserved rather than allocating real memory (WindowsNT/95, Solaris) they default to very large sizes (128MB), where real memory or space in the operating system swap file is taken immediately (SunOS), their default is very small (750KB,250KB).

Heap Memory Allocation

```
ec_set_option_long(EC_OPTION_PRIVATESIZE, 32*1024*1024);
ec_set_option_long(EC_OPTION_SHAREDSIZE, 64*1024*1024);
```

The sizes in bytes of the private and shared heaps. Normally these are ignored as the heaps grow as required.

They are used in the parallel ECL^{i}PS^e, since there allocation is done at fixed addresses and in that case these sizes determine the maximum amount of memory per heap.

Panic Function

```
        void my_panic(char * what, char * where);
...
ec_set_option_ptr(EC_OPTION_PANIC, my_panic);
```

When ECL^{i}PS^e experiences an unrecoverable error, this function is called. By default a function that prints the panic message and exits is called. After such an error, one should not call any of the functions in this interface.

Startup Arguments

```
        char *args[] = {"a","b","c"}
...
ec_set_option_long(EC_OPTION_ARGC, 3);
ec_set_option_ptr(EC_OPTION_ARGV, args);
```

These setting are used to simulate a command line for an embedded ECL^{i}PS^e, or to pass command line information to an embedded ECL^{i}PS^e. The ECL^{i}PS^e built-in predicates (`argc/1` and `argv/2`) can access this information. This provides a way of passing some initial data to the ECL^{i}PS^e engine.

Default Module

```
ec_set_option_ptr(EC_OPTION_DEFAULT_MODULE, "my_module");
```

The default module is the module in which goals called from the top-level execute. It is also the module that goals called from C or C++ execute in. The default setting is "eclipse".

I/O Initialisation

```
ec_set_option_long(EC_OPTION_IO, MEMORY_IO);
```

This option controls whether the default I/O streams of ECL^{i}PS^e are connected to stdin/stdout/stderr or to memory queues. The default setting of this option is SHARED_IO, which means the ECL^{i}PS^e streams 0,1,2 are connected to stdin/stdout/stderr. In an embedded application, stdin/stdout/stderr may not be available, or the host application may want to capture all I/O from ECL^{i}PS^e. In this case, use the MEMORY_IO setting, which creates queue streams for streams 0,1 and 2. These can then be read and written using ec_queue_read() and ec_queue_write().

Parallel system parameters

```
ec_set_option_long(EC_OPTION_PARALLEL_WORKER, 0);
ec_set_option_long(EC_OPTION_ALLOCATION, ALLOC_PRE);
ec_set_option_ptr(EC_OPTION_MAPFILE, NULL);
```

The above options are set differently in ECL^{i}PS^e when it is a worker in a parallel system. They should not be altered.

Appendix B

Summary of C++ Interface Functions

Note that apart from the methods and functions described here, all functions from the C interface which operate on simple types (int, long, char*) can also be used from C++ code.

B.1 Constructing ECL^{i}PS^e terms in C++

B.1.1 Class EC_atom and EC_functor

The ECL^{i}PS^e dictionary provides unique identifiers for name/arity pairs. EC_atoms are dictionary identifiers with zero arity, EC_functors are dictionary identifiers with non-zero arity.

EC_atom(char*)
: looks up or enters the given string into the ECL^{i}PS^e dictionary and returns a unique atom identifier for it.

char* EC_atom::name()
: returns the name string of the given atom identifier.

EC_functor(char*,int)
: looks up or enters the given string with arity into the ECL^{i}PS^e dictionary and returns a unique functor identifier for it.

char* EC_functor::name()
: returns the name string of the given functor identifier.

int EC_functor::arity()
: returns the arity of the given functor identifier.

B.1.2 Class EC_word

The EC_word is the basic type that all ECL^{i}PS^e data structures are built from (because within ECL^{i}PS^e typing is dynamic). The following are the functions for constructing ECL^{i}PS^e terms from the fundamental C++ types. CAUTION: constructed EC_words are only valid up to the next invocation of EC_resume(); afterwards they must be considered invalid. Using an invalid EC_word as an argument to any function of this interface may lead to a crash. The only thing

that can be done with an invalid EC_word is to assign a freshly constructed value to it. To preserve a value across invocations of EC_resume(), use EC_ref or EC_refs.

EC_word(const char *)
converts a C++ string to an ECL^{i}PS^e string. The string is copied.

EC_word(const int, const char *)
converts a C++ string of given length to an ECL^{i}PS^e string. The string is copied and can contain NUL bytes.

EC_word(const EC_atom)
creates an ECL^{i}PS^e atom from an atom identifier.

EC_word(const long)
converts a C++ long to an ECL^{i}PS^e integer.

EC_word(const long long)
converts a C++ long long to an ECL^{i}PS^e integer.

EC_word(const double)
converts a C++ double to an ECL^{i}PS^e double float.

EC_word(const EC_ref&)
retrieves the ECL^{i}PS^e term referenced by the EC_ref (see below).

EC_word term(const EC_functor,const EC_word args[ ])

EC_word term(const EC_functor,const EC_word arg1, ...)
creates an ECL^{i}PS^e compound term from the given components.

EC_word list(const EC_word hd, const EC_word tl)
Construct a single ECL^{i}PS^e list cell.

EC_word list(int n, const long*)
Construct an ECL^{i}PS^e list of length n from an array of long integers.

EC_word list(int n, const char*)
Construct an ECL^{i}PS^e list of length n from an array of chars.

EC_word list(int n, const double*)
Construct an ECL^{i}PS^e list of length n from an array of doubles.

EC_word array(int, const double*)
creates an ECL^{i}PS^e array (a structure with functor [ ]of appropriate arity) of doubles from the given C++ array. The data is copied.

EC_word matrix(int rows, int cols, const double*)
creates an ECL^{i}PS^e array (size rows) of arrays (size cols) of doubles from the given C++ array. The data is copied.

EC_word handle(const t_ext_type *cl, const t_ext_ptr data)
Construct an ECL^{i}PS^e handle for external data, attaching the given method table.

EC_word newvar()
Construct a fresh ECL^{i}PS^e variable.

EC_word nil()
Construct the empty list [ ].

B.2 Decomposing ECL^{i}PS^e terms in C++

The following methods type-check an ECL^{i}PS^e term and retrieve its contents if it is of the correct type. The return code is EC_succeed for successful conversion, an error code otherwise.

int EC_word::is_atom(EC_atom *)
checks whether the ECL^{i}PS^e pword is an atom, and if so, return its atom identifier.

int EC_word::is_string(char **)
checks whether the EC_word is a string (or atom) and converts it to a C++ string. This string is volatile, ie. it should be copied when it is required to survive resuming of ECL^{i}PS^e.

int EC_word::is_string(char **, long *)
checks whether the EC_word is a string (or atom) and converts it to a C++ string. This string is volatile, ie. it should be copied when it is required to survive resuming of ECL^{i}PS^e. The string's length is returned in the second argument.

int EC_word::is_long(long *)
checks whether the EC_word is an integer that fits into a C++ long, and if so, stores it via the pointer provided.

int EC_word::is_long_long(long long *)
checks whether the EC_word is an integer that fits into a C++ long long, and if so, stores it via the pointer provided.

int EC_word::is_double(double *)
checks whether the EC_word is a floating point number, and if so, returns it as an C++ double.

int EC_word::is_list(EC_word&,EC_word&)
checks whether the EC_word is a list and if so, returns its head and tail.

int EC_word::is_nil()
checks whether the EC_word is nil, the empty list.

int EC_word::functor(EC_functor *)
checks whether the EC_word is a compound term and if so, returns its functor.

int EC_word::arg(const int,EC_word&)
checks whether the EC_word is a compound term and if so, returns its nth argument.

int EC_word::arity()
returns the arity of an EC_word if it is a compound term, zero otherwise.

int EC_word::is_handle(const t_ext_type *, t_ext_ptr *)
checks whether the EC_word is a handle whose method table matches the given one, and if so, the data pointer is returned.

int EC_word::is_var()
checks whether the EC_word is a variable. Returns EC_succeed if so, EC_fail otherwise.

B.3 Referring to ECL^{i}PS^e terms from C++

The data types EC_refs and EC_ref provide a means to have non-volatile references to ECL^{i}PS^e data from within C++ data structures. However, it must be kept in mind that ECL^{i}PS^e data structures are nevertheless subject to backtracking, which means they may be reset to an earlier status when the search engine requires it. Creating a reference to a data structure does not change that in any way. In particular, when the search engine fails beyond the state where the reference was set up, the reference disappears and is also reset to its earlier value.

EC_refs(int n)
create a data structure capable of holding n non-volatile references to ECL^{i}PS^e data items. They are each initialised with a freshly created ECL^{i}PS^e variable.

EC_refs(int n,EC_word pw)
create a data structure capable of holding n non-volatile references to ECL^{i}PS^e data items. They are all initialised with the value pw, which must be of a simple type.

~EC_refs()
destroy the ECL^{i}PS^e references. It is important that this is done, otherwise the ECL^{i}PS^e garbage collector will not be able to free the references data structures, which may eventually lead to memory overflow.

EC_word EC_refs::operator[ ](int i)
return the ECL^{i}PS^e term referred to by the i'th reference.

void EC_refs::set(int i, EC_word new)
assign the term new to the i'th reference. This is a backtrackable operation very similar to **setarg/3**.

EC_word list(EC_refs&)
creates an ECL^{i}PS^e list containing all the elements of the EC_refs.

EC_ref()

EC_ref(EC_word pw)

~EC_ref()
analogous to the corresponding EC_refs constructors/destructor.

EC_ref& operator=(const EC_word)
assign a value to the EC_ref.

EC_word(const EC_ref&)
retrieves the ECL^{i}PS^e term referenced by the EC_ref.

B.4 Passing Data to and from External Predicates in C++

These two functions are only meaningful inside C++ functions that have been called from ECL^{i}PS^e as external predicates.

EC_word EC_arg(int i)
If inside a C++ function called from ECL^{i}PS^e, this returns the i'th argument of the call.

int unify(EC_word,EC_word)
Unify the two given pwords. Note that, if attributed variables are involved in the unification, the associated unification handlers as well as subsequent waking will only happen once control is returned to ECL^{i}PS^e.

int EC_word::unify(EC_word)
Similar, but a method of EC_word.

B.5 Operations on ECL^{i}PS^e Data

Interfaces to some basic operations on ECL^{i}PS^e Data.

int compare(const EC_word& pw1, const EC_word& pw2)
Similar to the compare/3 built-in predicate: returns 0 if the arguments are identical, a negative number if pw1 is smaller than pw2, and a positive number if pw1 is greater than pw2 in the standard term ordering.

int EC_word::schedule_suspensions(int)
Similar to the **schedule_suspensions/2** built-in predicate. Waking will only happen once control is returned to ECL^{i}PS^e and the wake/0 predicate is invoked. Return code is EC_succeed or an error code.

pword EC_word::free_handle(const t_ext_type*)
checks whether the EC_word is an ECL^{i}PS^e external data handle of the expected type, and calls its free-method. After doing that, the handle is stale and cannot be used any longer. Calling this method on an already stale handle silently succeeds. Return code is EC_succeed or an error code.

B.6 Initialising and Shutting Down the ECL^{i}PS^e Subsystem

These are the functions needed to embed ECL^{i}PS^e into a C++ main program.

int ec_init()
Initialise the ECL^{i}PS^e engine. This is required before any other functions of this interface can be used.

int ec_cleanup()
Shutdown the ECL^{i}PS^e engine.

B.7 Passing Control and Data to ECL^{i}PS^e from C++

These are the functions needed to embed ECL^{i}PS^e into C++ code.

void post_goal(const EC_word)

void post_goal(const char *)
post a goal (constraint) that will be executed when ECL^{i}PS^{e}is resumed.

int EC_resume(EC_word FromC, EC_ref& ToC)

int EC_resume(EC_word FromC)

int EC_resume()
resume execution of the ECL^{i}PS^e engine: All posted goals will be executed. The return value will be EC_succeed if the goals succeed (in this case the ToC argument returns a cut value that can be used to discard alternative solutions). EC_fail is returned if the goals fail, and EC_yield if control was yielded because of a yield/2 predicate call in the ECL^{i}PS^e code (in this case, ToC contains the data passed by the first argument of yield/2). If a writable queue stream with yield-option (see **open/4**) was flushed, the return value is PFLUSHIO and ToC contains the associated stream number. If there was an attempt to read from an empty queue stream with yield-option, the return value is PWAITIO and ToC contains the associated stream number. Moreover, if the previous EC_resume yielded due to a yield/2 predicate call, The term FromC is passed as input into the second argument of yield/2 on resuming.

void EC_ref::cut_to()
Should be applied to the ToC cut return value of an EC_resume(). Cut all choicepoints created by the batch of goals whose execution succeeded.

int post_event(EC_word Name)
Post an event to the ECL^{i}PS^e engine. This will lead to the execution of the corresponding event handler once the ECL^{i}PS^{e}execution is resumed. See also **event/1** and the User Manual chapter on event handling for more information. Name should be an ECL^{i}PS^e atom.

Appendix C

Summary of C Interface Functions

Note that a self-contained subset of the functions described here uses only integer and string arguments and is therefore suitable to be used in situations where no complex types can be passed, e.g. when interfacing to scripting languages.

C.1 Constructing ECL^{i}PS^e terms in C

All these functions return (volatile) pwords, which can be used as input to other constructor functions, or which can be stored in (non-volatile) ec_refs.

pword ec_string(const char*)
converts a C string to an ECL^{i}PS^e string. The string is copied.

pword ec_length_string(int, const char*)
converts a C string of given length to an ECL^{i}PS^e string. The string is copied and can contain NUL bytes.

pword ec_atom(const dident)
creates an ECL^{i}PS^e atom. A dident (dictionary identifier) can be obtained either via ec_did() or ec_get_atom().

pword ec_long(const long)
converts a C long to an ECL^{i}PS^e integer.

pword ec_long_long(const long long)
converts a C long long to an ECL^{i}PS^e integer.

pword ec_double(const double)
converts a C double to an ECL^{i}PS^e float.

pword ec_term(dident,pword,pword,...)
creates an ECL^{i}PS^e term from the given components.

pword ec_term_array(const dident,const pword[ ]**)**
creates an ECL^{i}PS^e term from the arguments given in the array.

pword ec_list(const pword,const pword)
creates a single ECL^{i}PS^e list cell with the given head (car) and tail (cdr).

pword ec_listofrefs(ec_refs)
creates an ECL^{i}PS^e list containing the elements of the ec_refs array.

pword ec_listoflong(int, const long*)
creates an ECL^{i}PS^e list of integers containing the longs in the given array. The data is copied.

pword ec_listofchar(int, const char*)
creates an ECL^{i}PS^e list of integers containing the chars in the given array. The data is copied.

pword ec_listofdouble(int, const double*)
creates an ECL^{i}PS^e list of doubles containing the doubles in the given array. The data is copied.

pword ec_arrayofdouble(int, const double*)
creates an ECL^{i}PS^e array (a structure with functor [ ]of appropriate arity) of doubles from the given C array. The data is copied.

pword ec_matrixofdouble(int rows, int cols, const double*)
creates an ECL^{i}PS^e array (size rows) of arrays (size cols) of doubles from the given C array. The data is copied.

pword ec_handle(const t_ext_type*,const t_ext_ptr)
creates an ECL^{i}PS^e handle that refers to the given C data and its method table.

pword ec_newvar()
creates a fresh ECL^{i}PS^e variable.

pword ec_nil()
creates an ECL^{i}PS^e nil ie. the empty list [ ].

Auxiliary functions to access the ECL^{i}PS^e dictionary.

dident ec_did(char*,int)
looks up or enters the given string with arity into the ECL^{i}PS^e dictionary and returns a unique identifier for it.

char* DidName(dident)
returns the name string of the given dictionary identifier.

int DidArity(dident)
returns the arity of the given dictionary identifier.

C.2 Decomposing ECL^{i}PS^e terms in C

The following group of functions type-check an ECL^{i}PS^e term and retrieve its contents if it is of the correct type. The return code is PSUCCEED for successful conversion. If a variable was encountered instead INSTANTIATION_FAULT is returned. Other unexpected types yield a TYPE_ERROR. Special cases are explained below.

int ec_get_string(const pword,char)**
checks whether the ECL^{i}PS^e pword is a string (or atom) and converts it to a C string. This string is volatile, ie. it should be copied when it is required to survive resuming of ECL^{i}PS^e.

int ec_get_string_length(const pword,char,long*)**
the same as ec_get_string(), but returns also the string's length. Note that ECL^{i}PS^e strings may contain null characters!

int ec_get_atom(const pword,dident*)
checks whether the ECL^{i}PS^e pword is an atom, and if so, return its dictionary identifier.

int ec_get_long(const pword,long*)
checks whether the ECL^{i}PS^e pword is an integer that fits into a C long, and if so, stores it via the pointer provided. Otherwise, the return code is RANGE_ERROR.

int ec_get_long_long(const pword,long long*)
checks whether the ECL^{i}PS^e pword is an integer that fits into a C long long, and if so, stores it via the pointer provided. Otherwise, the return code is RANGE_ERROR.

int ec_get_double(const pword,double*)
checks whether the ECL^{i}PS^e pword is a floating point number, and if so, returns it as an C double.

int ec_get_nil(const pword)
checks whether the ECL^{i}PS^e pword is nil, the empty list.

int ec_get_list(const pword,pword*,pword*)
checks whether the ECL^{i}PS^e pword is a list, and if so, returns its head and tail. If it is nil, the return code is PFAIL.

int ec_get_functor(pword,dident*)
checks whether the ECL^{i}PS^e pword is a structure, and if so, returns the functor.

int ec_get_arg(const int n,pword,pword*)
checks whether the ECL^{i}PS^e pword is a structure, and if so, returns the n'th argument. The return code is RANGE_ERROR if the argument index is out of range.

int ec_arity(pword)
returns the arity (number of arguments) of an ECL^{i}PS^e pword if it is a structure, otherwise zero.

int ec_get_handle(const pword,const t_ext_type*,t_ext_ptr*)
checks whether the ECL^{i}PS^e pword is a handle whose method table matches the given one, and if so, the data pointer is returned.

int ec_is_var(const pword)
checks whether the ECL^{i}PS^e pword is a variable. Note that the return values are PSUCCEED are PFAIL rather than standard C truth values.

C.3 Referring to ECL^{i}PS^e terms from C

The data types ec_refs and ec_ref provide a means to have non-volatile references to ECL^{i}PS^e data from within C data structures. However, it must be kept in mind that ECL^{i}PS^e data structures are nevertheless subject to backtracking, which means they may be reset to an earlier status when the search engine requires it. Creating a reference to a data structure does not change that in any way. In particular, when the search engine fails beyond the state where the reference was set up, the reference disappears and is also reset to its earlier value.

ec_refs ec_refs_create(int n,const pword pw)
create a data structure capable of holding n non-volatile references to ECL^{i}PS^e data items. They are initialised with the value pw, which must be of a simple type.

ec_refs ec_refs_create_newvars(int)
like ec_refs_create(), but each item is initialised to a freshly created ECL^{i}PS^e variable.

void ec_refs_destroy(ec_refs)
destroy the ECL^{i}PS^e references. It is important that this is done, otherwise the ECL^{i}PS^e garbage collector will not be able to free the references data structures, which may eventually lead to memory overflow.

void ec_refs_set(ec_refs,int i,const pword pw)
set the i'th reference to the ECL^{i}PS^e term pw. This setting is subject to the ECL^{i}PS^e engine's undo-mechanism on backtracking.

pword ec_refs_get(const ec_refs,int i)
return the ECL^{i}PS^e term referred to by the i'th reference.

int ec_refs_size(const ec_refs)
return the capacity of the ec_refs data structure.

ec_ref ec_ref_create(pword)
like ec_refs_create() for a single reference.

ec_ref ec_ref_create_newvar()
analogous to ec_refs_create_newvars().

void ec_ref_destroy(ec_ref)
analogous to ec_refs_destroy().

void ec_ref_set(ec_ref,const pword)
analogous to ec_refs_set().

pword ec_ref_get(const ec_ref)
analogous to ec_refs_get().

C.4 Passing Data to and from External Predicates in C

These two functions are only meaningful inside C functions that have been called from ECL^{i}PS^e as external predicates.

pword ec_arg(int i)
If inside a C function called from ECL^{i}PS^e, this returns the i'th argument of the call.

int ec_unify(pword,pword)
Unify the two given pwords. Note that, if attributed variables are involved in the unification, the associated unification handlers as well as subsequent waking will only happen once control is returned to ECL^{i}PS^e.

C.5 Operations on ECL^{i}PS^e Data

Interfaces to some basic operations on ECL^{i}PS^e Data.

int ec_compare(const pword pw1, const pword pw2)
Similar to the compare/3 built-in predicate: returns 0 if the arguments are identical, a negative number if pw1 is smaller than pw2, and a positive number if pw1 is greater than pw2 in the standard term ordering.

int ec_schedule_suspensions(pword,int)
Similar to the **schedule_suspensions/2** built-in predicate. Waking will only happen once control is returned to ECL^{i}PS^e and the wake/0 predicate is invoked. Return code is PSUCCEED or an error code.

int ec_free_handle(const pword, const t_ext_type*)
checks whether pw is an ECL^{i}PS^e external data handle of the expected type, and calls its free-method. After doing that, the handle is stale and cannot be used any longer. Calling this function on an already stale handle silently succeeds. Return code is PSUCCEED or an error code.

C.6 Initialising and Shutting Down the ECL^{i}PS^e Subsystem

These are the functions needed to embed ECL^{i}PS^e into a C main program.

int ec_set_option_long(int, long)
Set the value of a numerical option (see appendix A). The numerical value can be of type long.

int ec_set_option_ptr(int, char *)
Set the value of a string-valued option (see appendix A).

int ec_init()
Initialise the ECL^{i}PS^e engine. This is required before any other functions of this interface (except option setting) can be used.

int ec_cleanup()
Shutdown the ECL^{i}PS^e engine.

C.7 Creating External Predicates in C

This function serves the same purpose as the ECL^{i}PS^e built-in external/2:

int ec_external(dident pred, int(*fct)(), dident module)
Creates a predicate pred in the given module, whose C/C++ implementation is the function fct (see chapter 4 for how to write such functions). The module must exist. Return code is PSUCCEED or an error code.

C.8 Passing Control and Data to ECL^{i}PS^e from C

These are the functions needed to embed ECL^{i}PS^e into C code.

void ec_post_goal(const pword)
post a goal (constraint) that will be executed when ECL^{i}PS^{e}is resumed.

void ec_post_string(const char *)
same as ec_post_goal(), but the goal is given in ECL^{i}PS^e syntax in a string. This should only be used if speed is not critical and if the goal does not contain variables whose values may be needed later. This function is part of the simplified interface.

void ec_post_exdr(int len, const char *exdr)
same as ec_post_goal(), but the goal is given in EXDR format (see chapter 9). This function is part of the simplified interface.

int ec_resume()
resume execution of the ECL^{i}PS^e engine: All posted goals will be executed and all posted events will be handled. The return value will be PSUCCEED if the goals succeed PFAIL is returned if the goals fail, and PYIELD if control was yielded because of a **yield/2** predicate call in the ECL^{i}PS^e code. If a writable queue stream with yield-option (see **open/4**) was flushed, the return value is PFLUSHIO. If there was an attempt to read from an empty queue stream with yield-option, the return value is PWAITIO. If an asynchronous ECL^{i}PS^e thread is already running, PRUNNING is returned. No parameters can be passed. This function is part of the simplified interface.

int ec_resume1(ec_ref ToC)
Similar to ec_resume(), but if the return value is PSUCCED, the ToC argument returns a cut value that can be used to discard alternative solutions by passing it to ec_cut_to_chp(). If the return value is PYIELD, control was yielded because of a **yield/2** predicate call in the ECL^{i}PS^e code, and ToC contains the data passed by the first argument of **yield/2**. If the return value is PFLUSHIO or PWAITIO, ToC contains the associated stream number.

int ec_resume2(const pword FromC,ec_ref ToC)
Similar to ec_resume1(), but it allows to pass an argument to the resumed execution. This is only useful if the execution had yielded due to a **yield/2** predicate call. The term FromC is passed as input into the second argument of **yield/2**.

int ec_resume_long(long *ToC)
Similar to ec_resume1(), but allows only integer values to be passed from ECL^{i}PS^e to C (otherwise TYPE_ERROR is returned). This function is part of the simplified interface.

int ec_resume_async()
Similar to ec_resume(), but ECL^{i}PS^e is resumed in a separate thread in case this is supported by the operating system. The return value is PSUCCED if the thread started successfully, SYS_ERROR if there was a problem creating the thread, and PRUNNING if there was already an ECL^{i}PS^e thread running (only one ECL^{i}PS^e thread is allowed to run at any time). If threads are not supported, the call does nothing and return PSUCCED. Use ec_resume_status() to wait for termination and to retrieve the results of the execution.

int ec_resume_status()
This function is supposed to be called after a call to ec_resume_async(). It returns PRUNNING as long as the ECL^{i}PS^e thread is still running. If the thread has stopped, the return values are the same as for ec_resume(). If threads are not supported, the pair of ec_resume_async() and ec_resume_status() is equivalent to an ec_resume().

int ec_resume_status_long(long *ToC)
Similar to ec_resume_status(), but allows an integer to be returned to the caller, as done by ec_resume_long().

int ec_wait_resume_status_long(long *ToC, int timeout)
Similar to ec_resume_status_long(), but waits for the ECL^{i}PS^e thread to finish execution. The function returns as soon as the ECL^{i}PS^e thread is finished, or after timeout milliseconds, whatever is earlier. In case of timeout, the return value will be PRUNNING. If timeout is zero, the function is equivalent to ec_resume_status_long(). If timeout is negative, there will be no timeout and the function will only return when the ECL^{i}PS^e thread is finished.

int ec_handle_events(long *ToC)
Similar to ec_resume_long(), but posted goals are not executed, only events are handled.

void ec_cut_to_chp(ec_ref)
Cut all choicepoints created by the batch of goals whose execution succeeded. The argument should have been obtained by a call to ec_resume2().

int ec_post_event(pword Name)
Post an event to the ECL^{i}PS^e engine. This will lead to the execution of the corresponding event handler once the ECL^{i}PS^{e}execution is resumed. See also **event/1** and the User Manual chapter on event handling for more information. Name should be an ECL^{i}PS^e atom.

int ec_post_event_string(const char *)
Post an event to the ECL^{i}PS^e engine. This will lead to the execution of the corresponding event handler once the ECL^{i}PS^{e}execution is resumed. See also **event/1** and the User Manual chapter on event handling for more information. The event name is given as a string. This function is part of the simplified interface.

C.9 Communication via ECL^{i}PS^e Streams

These functions allow exchanging data with an embedded ECL^{i}PS^e via queue streams created by the ECL^{i}PS^e code. Queue streams can be created either by using **open/3** and **open/4** from

within ECL^{i}PS^e code, or by initializing ECL^{i}PS^e with the MEMORY_IO option. In the latter case, the streams 0, 1 and 2 are queues corresponding to ECL^{i}PS^e's input, output and error streams.

int ec_queue_write(int stream, char *data, int size)
Write string data into the specified ECL^{i}PS^e queue stream. Data points to the data and size is the number of bytes to write. The return value is 0 for success, or a negative error number.

int ec_queue_read(int stream, char *buf, int size)
Read string data into the specified ECL^{i}PS^e queue stream. Buf points to a data buffer and size is the buffer size. The return value is either a negative error code, or the number of bytes read into buffer.

int ec_queue_avail(int stream)
Determines the number of bytes that are available and can be read from the given queue stream. The return value is either that number or a negative error code.

int ec_stream_nr(char *name)
Get the stream number of the named stream. If the return value is negative then there is no open stream with the specified name. This is the same operation that the ECL^{i}PS^e built-in **get_stream/2** performs).

C.10 Miscellaneous

These two functions provide an alternative method for posting goals and retrieving results. They are intended for applications with a simple structure that require only infrequent call-return style control transfers and little information passing between ECL^{i}PS^e and C. It is less powerful and less efficient that the primitives described above.

int ec_exec_string(char*,ec_ref Vars)
let ECL^{i}PS^e execute a goal given in a string ECL^{i}PS^e syntax. Return value is PSUCCEED or PFAIL, depending on the result of the execution. If successful, Vars holds a list mapping the variables names within the string to their values after execution.

int ec_var_lookup(ec_ref Vars,char*,pword* pw)
Lookup the value of the variable with the given name. Vars is a list as returned by ec_exec_string().

Appendix D

Foreign C Interface

This library (loaded with `:- lib(foreign)`) allows to use external functions written for Quintus or SICStus Prolog, or to interface C functions which are independent of ECL^{i}PS^e. It accepts the syntax and semantics of the predicates **foreign/3**, **load_foreign_files/2** and **load_foreign_files/2** of Quintus/SICStus. Since their external interface is incompatible with the ECL^{i}PS^e one, this library generates a C source file which converts the ECL^{i}PS^{e}interface into the foreign one, by defining a new C function for every C function defined in the foreign interface. Note that this approach uses more C code, but it is still more efficient than using a generic procedure to check the argument of every external function.
After compiling definitions of the **foreign/3** predicate, (the predicate **foreign_file/2** is ignored), the predicate **make_simple_interface** has to be called. This predicate generates a file *interface.c*, which contains the auxiliary C definitions. This file has to be compiled to obtain the *interface.o* file, it might also have to be edited to include other *.h* files. After the file *interface.o* has been generated, the system is fully compatible with the Quintus/SICStus foreign interface, and calling **load_foreign_files/2** will connect the external functions with ECL^{i}PS^e.
Although it is possible to use this library to interface existing independent C functions, its main use is to port foreign interface from Quintus or SICStus. Please refer to their manuals for the description of the foreign interface.

Index