ECL^{i}PS^e

Embedding and Interfacing Manual

Release 4.2

Stefano Novello (IC-Parc)
Joachim Schimpf (IC-Parc)

August 6, 1999

Trademarks

Contents

Chapter 1

Introduction

This manual contains the information needed to interface ECL^{i}PS^e code to C or C++ environments, or to use it from within scripting languages. 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.
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 pre-requisites 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 Static linking (Unix/Linux)

If your operating system only supports static linking, or if you want to link statically for some reason, you have to link explicitly with `libeclipse.a` and the necessary support libraries must be specified, e.g.

```
gcc -I$ECLIPSEDIR/include/$ARCH eg_c_basic.c $ECLIPSEDIR/lib/$ARCH/libeclipse.a
    -L$ECLIPSEDIR/lib/$ARCH -lgmp -lshm -ldummies -ldl -lnsl -lsocket -lm
```

The libraries `gmp`, `shm` and `dummies` are ECL^{i}PS^e support libraries and must be specified in that order. The others are Unix libraries.
It is recommended that you copy the makefile "Makefile.external" provided in your installation directory under lib/$ARCH and adapt it for your purposes.

2.1.5 Compiling, linking and running on Windows

In the **Link** section of the compiler/development system's settings, specify `eclipse.lib` as an additional library, and the location of this library, e.g. `C:/Eclipse/lib/i386_nt` as an additional library path.
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 -1 if an error occurred. It is possible to influence the initialisation of ECL^{i}PS^e by setting some fields in a structure called `ec_options`. These fields and their purpose are 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 */
ec_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 will be garbage collected or moved so terms do not survive the execution of ECL^{i}PS^e. In particular, one cannot build such terms asynchronously while ECL^{i}PS^e is running, for example this precludes building terms from within a signal handler unless it can make sure that ECL^{i}PS^e has yielded when it is running.

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

It is possible to simply 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. In C++ you can just cast.

```
/* 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.

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";
```

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;
```

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 **erase_array/1** or **abolish_record/1**. 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).

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).

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.

string_size(t_ext_ptr obj, int quoted)

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.

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.

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.

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

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

Example of the simplest possible user-defined method table:

```
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 -c -I/usr/local/eclipse/include/sparc_sunos5
           -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.

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 use ECL^{i}PS^e from within 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.
The **tkeclipse** development environment is entirely implemented using the facilities described in this chapter.

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
```

It might also be necessary to provide information about where the DLLs or shared library files can be found. On Unix systems this is done by setting the LD_LIBRARY_PATH environment variable, e.g.

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/location/of/my/eclipse/lib/sparc_sunos5
```

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 Chapter A for their meaning.

ec_init
: Initialise the ECL^{i}PS^e engine. This is required before any other commands of this interface (except ec_set_option) can be used.

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 follow 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 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 first posting it and then transferring control via the **ec_resume** command. The related commands are the following:

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 5.6. 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 5.5.2).

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_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.

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 realiably is **ec_post_event**.

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 5.6.

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

5.4 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. To create a communication channel between ECL^{i}PS^e and Tcl, 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_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_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.

5.4.1 From ECL^{i}PS^e to Tcl

To create a queue from ECL^{i}PS^e to Tcl, first create a write-queue in ECL^{i}PS^e, then use the queue name to open this queue as a Tcl channel in read mode, e.g.

```
ECLiPSe:        open(queue(""), write, my_out_queue).
Tcl:            set my_in_channel [ec_queue_connect my_out_queue r]
```

Now the queue 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).
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 5.6) 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 5.6) on the Tcl side:

```
ECLiPSe:        write_exdr(my_out_queue, foo(bar,3)).
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 5.6.

5.4.2 From Tcl to ECL^{i}PS^e

To create a queue from Tcl to ECL^{i}PS^e, first create a read-queue in ECL^{i}PS^e, then use the queue name to open this queue as a Tcl channel in write mode:

```
ECLiPSe:        open(queue(""), read, my_in_queue)
Tcl:            set my_out_channel [ec_queue_connect my_in_queue w]
```

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
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 5.6) format for communication. This allows to send and receive structured and typed data. The primitives to do that are ec_read_exdr (section 5.6) 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)
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 5.6.

5.5 Attaching Handlers to Queues

5.5.1 ECL^{i}PS^e to Tcl

In order to handle ECL^{i}PS^e I/O on queues more conveniently, it is possible to associate a Tcl handler with every queue. These handlers can be invoked automatically whenever ECL^{i}PS^e flushes a write-queue or reads from an empty read-queue.
For that purpose, the queue must be created using **open/4** with the yield-option. 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:

```
ECLiPSe:  open(queue(""), write, my_out_queue, [yield(on)]).
Tcl:      pack [text .tout]
          ec_queue_connect my_out_queue r {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.
Similarly, an ECL^{i}PS^e input queue could be configured as follows:

```
ECLiPSe:  open(queue(""), read, my_in_queue, [yield(on)]).
Tcl:      ec_queue_connect my_in_queue w \
                           {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_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.

5.5.2 Tcl to ECL^{i}PS^e

A queue whose read-end is on the ECL^{i}PS^e-side 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 using **open/4** with the event-option, e.g.[1]

```
open(queue(""), read, my_queue, [event(my_queue_event)])
```

Assuming something was written into the queue from within Tcl code, the ECL^{i}PS^e event will be handled as soon as ECL^{i}PS^e is resumed or **ec_handle_events** (a restricted form of **ec_resume**) is invoked.
Note that it is also possible to raise ECL^{i}PS^e events which are not linked to queues, using **ec_post_event**.

[1] 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.

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

EXDR (ECL^{i}PS^e External Data Representation, see also chapter 7) 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:

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
Integer	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
```

Chapter 6

Embedding into Visual Basic

This is a set of Visual Basic classes built around the scripting language interface of ECL^{i}PS^e. They are provided in source form in directory doc/examples/vb within the installation. The interface is similar in spirit to the ECL^{i}PS^e embedding interfaces for other languages.

6.1 The EclipseThread Project

This contains the classes which form the interface to eclipse.

class EclipseClass
An object of this class is a thread running eclipse code. Only one such object may exist in a process.

class EclipseStreams
A collection of queue streams for communicating with ECL^{i}PS^e.

class EclipseStream
A stream of on which data can be sent to or from ECL^{i}PS^e.

6.2 Public Enumerations

Enum EC_Status
Symbolic names for the status values returned by goal execution.

Enum EclipseStreamMode
Symbolic names for the direction in which data flows within an EclipseStream.

Enum EC_ERROR
Symbolic names for error conditions in the interface.

6.3 The EclipseClass class

An object of this class is an entity running eclipse code. Only one such object may exist in a process.
The class provides methods to execute goals and to access queue streams to communicate with the running goal.

Function Init() As Long

Initialise the ECL^{i}PS^e engine. This is required before any other functions of this interface can be used.

Sub Send(EventName As String)

Post an event to the ECL^{i}PS^e engine. This will lead to the execution of the corresponding event handler See also **event/1** and the User Manual chapter on event handling for more information. The event name is given as a string. Note that if ECL^{i}PS^e was not running, the event stays in its queue until it is resumed.

Function Post(Goal As String) As EC_Status

Post a goal (constraint) that will be executed when ECL^{i}PS^{e}is resumed. The goal is given as a string in ECL^{i}PS^e syntax.

Function ResumeAsync() As EC_Status

Resume execution of the ECL^{i}PS^e engine: All posted goals will be executed. The return value will be 'Success' if the goals succeed 'Fail' is returned if the goals fail, and 'Yield' if control was yielded because of a **yield/2** predicate call in the ECL^{i}PS^e code. No parameters can be passed.

The function returns when the posted goals have finished executing. Since a separate thread is actually executing the goals though, events may be received by the Visual Basic program during the execution of this function. It is an error to call this function recursively while handling one of these events.

Function HandleEvents() As EC_Status

Resume execution of the ECL^{i}PS^e engine, but do not let it execute any posted goals. Only ECL^{i}PS^e events will be handled. Sources of such events are the Post() Function or writing to an event-raising ECL^{i}PS^e queue stream. The function returns when the events have all been handled by ECL^{i}PS^e and the return value is 'Success'. It is an error to call this function while a ResumeAsync() is still active.

Sub RPC(Goal As Variant, Response As Variant)

ECL^{i}PS^e Remote Predicate Call. An ECL^{i}PS^e goal is constructed from *Goal* according to the conversion rules explained in chapter 7. Goal is called 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. On success, the (possibly more instantiated) Goal is returned as Response, otherwise "fail" or "throw" respectively.

Unlike **ResumeAsync**, calls to **RPC** can be nested and can be used from within VB Stream event handlers.

Property Let EclipseDir(Dir As String)

The directory where ECL^{i}PS^e is installed. See Chapter A.

Property Let Module(Mod As String)

The default module for calling goals. See Chapter A.

Property Let GlobalSize(Size As Long)

The maximum size of the ECL^{i}PS^e global/trail stack in bytes. See Chapter A.

Property Let LocalSize(Size As Long)
The maximum size of the ECL^{i}PS^e local/control stack in bytes. See Chapter A.

Property Let SharedSize(Size As Long)
The maximum size of the ECL^{i}PS^e shared heap. See Chapter A.

Property Let PrivateSize(Size As Long)
The maximum size of the ECL^{i}PS^e private heap. See Chapter A.

Property Get Streams
The EclipseStreams collection associated with this EclipseClass.

6.4 The EclipseStreams Collection Class

This is a collection of EclipseStream objects. The keys to this collection are the symbolic name of ECL^{i}PS^e streams. Initially it will contain the 'input' 'output' and 'error' streams.

Function Add(Key As String, Mode As EclipseStreamMode) As EclipseStream
Create a new EclipseStream. 'Key' must be the symbolic name of an existing ECL^{i}PS^e queue stream. These are created using the **open/3** or **open/4** built-in. If 'Mode' is 'FromEclipse' the ECL^{i}PS^e stream must have been opened in 'write' mode. If it is 'ToEclipse' the ECL^{i}PS^{e}stream must have been opened in read mode.

Property Get Item(vntIndexKey As Variant) As EclipseStream
Used to retrieve streams from the collection. 'vntIndexKey' can be either the symbolic steam name or an integer index into the collection.

Property Get Count() As Long
The number of items in the collection

Sub Remove(vntIndexKey As Variant)
Remove an EclipseStream from the collection. (This does not close the corresponding ECL^{i}PS^e stream though).

6.5 The EclipseStream Class

This class allows exchanging data with an embedded ECL^{i}PS^e via queue streams created by the ECL^{i}PS^e code.

Event Flush
Raised whenever the ECL^{i}PS^e program flushes this stream.

Property Get Key() As String
The symbolic name of this stream

Property Get Mode() As EclipseStreamMode
The direction in which data is sent over this EclipseStream

Property Get/Let Prompt() As String
A prompt string. This appears in an input box that pops up when the ECL^{i}PS^e program attempts to read from a queue stream if no data is available.

Sub StreamWrite(Data As String)
Send 'Data' over this stream.

Function Read(l As Long) As String
Receives at most 'l' characters from the EclipseStream. No flushing is necessary.

Function NewData() As String
Receives all available characters from the EclipseStream that has been written on the stream since the last flush.

Sub WriteExdr(Data As Variant)
Writes the given data structure onto the stream in EXDR-encoded form. See chapter 7 for details about EXDR format.

Sub ReadExdr(Data As Variant)
Reads one EXDR-encoded term from the stream and returns its VB-representation in Data. See chapter 7 for details about EXDR format.

Chapter 7

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 as follows:

EXDR type	**ECLiPSe type**	**TCL type**	**VB type**
Integer	integer	int	Long
e.g.	123	123	123
Double	real	double	Double
e.g.	12.3	12.3	12.3
String	string	string	String
e.g.	"abc"	abc	"abc"
List	./2	list	Collection of Variant
e.g.	[a,b,c]	{a b c}	
Nil	[ ]/0	empty string	Empty Collection of Variant
e.g.	[ ]	{} ""	
Struct	compound	list	Array of Variant
e.g.	foo(bar,3)	{foo bar 3}	
Variable	variable	string	Empty Variant
e.g.	_	_	Empty

The EXDR Integer data type is a 32-bit integer, therefore 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.

7.1 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 6 for the VB primitives for manipulating EXDR terms.

7.2 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_VERION 1:

```
Term            ::=     'V' Version (Integer|Double|String|List
                                     |Nil|Struct|Variable)
Integer         ::=     'I' XDR_int
Double          ::=     'D' XDR_double
String          ::=     'S' Length <byte>*
List            ::=     '[' Term (List|Nil)
Nil             ::=     ']'
Struct          ::=     'F' Arity String Term*
Variable        ::=     '_'
Length          ::=     XDR_int                         // >= 0
Arity           ::=     XDR_int                         // >= 0
Version         ::=     <byte>
XDR_int         ::=     <4 bytes, msb first>
XDR_double      ::=     <8 bytes, ieee double, exponent first>
```

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_int() 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_int(EC_OPTION_LOCALSIZE, 128*1024*1024);
ec_set_option_int(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_int(EC_OPTION_PRIVATESIZE, 32*1024*1024);
ec_set_option_int(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_int(EC_OPTION_ARGC, 3);
ec_set_option_ptr(EC_OPTION_ARGV, args);
```

ECL^{i}PS^e has two built-in predicates (`argc/1` and `argv/2`) for looking at its command line arguments. These look at the `Argc` and `Argv` fields of the `ec_options` structure so 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_int(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_int(EC_OPTION_PARALLEL_WORKER, 0);
ec_set_option_int(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:

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 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 a word-sized integer, and if so, returns it as a C++ long.

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_list(EC_word&,EC_word&)
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.

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.

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.

B.5 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.6 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_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 a word-sized integer, and if so, returns it as a C long.

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.

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.

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.

C.5 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_int(int, int)
Set the value of a numerical option (see appendix A).

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.6 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 7). 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_long(), but allows an integer to be returned to the caller, as done by ec_resume_long().

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.7 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_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.8 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**, **foreign_file/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