4.900

Intelligence Server

by Alan Westwood

WIN-PROLOG Intelligence Server

Written by Alan Westwood

Logic Programming Associates Ltd
Studio 30, Royal Victoria Patriotic Building
Trinity Road, London SW18 3SX, England

Phone: +44 (0) 20 8871 2016
Fax: +44 (0) 20 8874 0449

Info: info_team@lpa.co.uk
Sales: sale_team@lpa.co.uk
Support: tech_team@lpa.co.uk
Web: http://www.lpa.co.uk

Table of Contents

Chapter 1 Intelligence Server

The *WIN-PROLOG* "Intelligence Server" provides a full-featured Prolog server for applications written using almost any Windows programming language or visual development system. It not only lets applications use the inferencing power of Prolog in the background, but also offers 'call-back' facilities which allow the Prolog code to request any additional input needed to complete its computations.

Getting started

For those who prefer a hands on approach to getting to know the ropes, you could do worse than start with the section: Testing the Prolog code in WIN-PROLOG via the Intelligence Server in Chapter 10 Programming using the Intelligence Server. This will allow you to experiment with the Intelligence Server within the sandbox of an interactive *WIN-PROLOG* session.

The features of the Intelligence Server

The Intelligence Server interface provides a number of features that directly benefit the development of distributed intelligent applications, these can be summarised as follows:

Text-based interface you can use this clean and safe interface between languages with radically different data types.

Language independence you can develop the Prolog component independently of the language used to build the client component.

Extendibility you can use any Windows language that supports 32-bit DLL calls as the client.

Prolog backtracking each solution to a multiple solution Prolog goal is returned individually.

Clean and safe development because the Prolog code for the server utilises pure text input and output, you can easily develop it in isolation. This means that your Prolog code can be fully tested and debugged using the extensive features of the *WIN-PROLOG* development environment.

Callbacks these allow a Prolog goal to request additional information from your client code and then to continue with the computation.

A safe and simple text interface

A general problem associated with mixed language programming is that the languages tend to come from completely different worlds and use very different internal data structures. In the Prolog world not only are the data structures complex, they are also dynamic. This means that Prolog requires the use of garbage collection to tidy up memory once such a structure is no longer in use. Any system that allowed arbitrary Prolog terms to be built piecemeal would also have to take garbage collection into account.

Rather than provide direct mappings between the complex structures of Prolog and a multiplicity of other languages, The *Intelligence Server* provides a uniform and safe way of communicating via a simple text interface. In this model, the client application sends queries to Prolog, and receives back any output performed by the user's Prolog code as text strings.

To facilitate this text interface, ***WIN-PROLOG*** has a special string data type, which is optimised for inputting and outputting significant amounts of text data. This string data type is also backed up by a number of versatile text formatting predicates. In each function call the Intelligence Server can send as much as 64k, this corresponds to quite a significant Prolog goal.

Suppose that an application wanted to call ***WIN-PROLOG*** 's *member/2* predicate to return the individual elements of a list. If knowledge of Prolog's internal mechanism for constructing lists were required, constructing the call to *member/2* would be a complex process. The application would need to call several functions; to build the list, to build each element of the list, create a variable, get the variable binding on return, and so on.

In the Intelligence Server model you can construct a text string that represents the Prolog term and then pass this to Prolog. This is a much safer approach. If you made any mistake in creating the text of the term, all that would happen is that a syntax error would be returned, rather than a GPF!

Prolog independence from the client language

Because the Intelligence Server uses text as the medium for the interface, standard Prolog code can be developed and then integrated into any language required. You can develop your application's front-end in Visual Basic 6 and then replace it with C++ or Java as and when the need requires without having to alter your Prolog code at all.

Extendibility

The Intelligence Server comprises six function calls defined in the library *INT386W.DLL*. Through this interface any Windows language supporting 32-bit DLL function calls can be used as a client language.

There is only one DLL which implements the *Intelligence Server* interface, namely *INT386W.DLL*, all languages use this DLL, however, there are different support DLL's that manage the interface between this DLL and the front-end. Delphi, C and C++ for example can use the DLL directly. VB6, .NET and Java however require an intermediate DLL. Python has a simple module that manages the DLL, Int386w.py .

Backtracking

Backtracking is fully supported in the *Intelligence Server.* When a Prolog predicate is called and returns a result, it maintains its current backtracking position, so that each successive call returns the next available solution. It is analogous to typing a goal at the Prolog prompt and getting each successive solution on backtracking by pressing the space bar.

Clean and safe development

In any multi-language application, the safe testing and debugging of the individual components becomes an important issue. In a system where direct callbacks are involved, where one language can directly call functions in the other, this testing becomes extremely complex, as the development environments for all the languages involved need to be running at the same time. Because the Prolog code in an Intelligence Server application uses pure text as its input and output, it can be easily tested and debugged in isolation from the client code. The ***WIN-PROLOG*** development environment provides a number of support features including a sophisticated source-level debugger.

Callbacks

An essential feature for a client-server interface is to allow the server to request additional information from the client during a computation. To permit this in the *Intelligence Server* the predicate, *input/2* is defined. When *input/2* is called as part of the Prolog server code, the predicate making the call is

suspended until the client returns the requested data, at which point the predicate can continue its computation. Typically, this is a requirement for expert system applications where the inference engine requires additional data to reach a solution.

Unicode

From version 4.100 the Intelligence Server supports the passing of Unicode 16-bit strings to and from the language of your choice. The Ansi 8-bit string interface is still supported, and indeed the two can be mixed. All that is required to change the Prolog mode of the interface is a single parameter to the function LoadProlog. Depending upon the front-end application language more work may be required to support Unicode strings. For example, in the case of Java little or no work is needed as Java fully supports Unicode strings in its String type. In C or Delphi however strings that are required to be Unicode need to be defined as such. Please see your language documentation for more details.

Buffer Size

From version 4.100 the size of the internal buffer used by the Intelligence Server can be specified. A default value of 64Kb will be used if a size of 0 is specified.

Version 4.320 introduced the Tickle argument to the LoadProlog function.

Only one function has changed to support the above, namely LoadProlog which now has the following specification:

```
UINT LoadProlog( LPSTR CommandLine, UINT BufferSize, UINT Encoding, UINT
Tickle )
```

Where:

CommandLine	An ansi (ISO/IEC 8859-1) string specifying the prolog command line.
BufferSize	An integer specifying the size of the internal buffer, a value of 0 specifies a default value of 64Kb.
Encoding	Specifies the encoding of strings passed to and from Prolog: 0: Ansi (ISO/IEC 8859-1) 0 terminated strings 1: Unicode 16-bit characters.
Tickle	Specifies whether Prolog should be tickled every so often to keep the interface high up the windows priority queue, this can significantly increase the speed of the interface: 0: Don t tickle 1: Do Tickle

These changes mean that if you use the call LoadProlog("ansi string", 0, 0, 0) then code will work unchanged from previous versions of the Intelligence Server.

The structure of an Intelligence Server application

To show more clearly how the Intelligence Server works, let s examine the components that go to make up the interface. An Intelligence Server application can be divided into three main parts:

The client is your application code that is requesting the services of the *Intelligence Server*, and may be written in any language that supports 32-bit DLL function calls.

The ***WIN-PROLOG*** application is your Prolog code that serves requests from the client.

The Intelligence Server interface. This enables the interface between the client and the ***WIN-PROLOG*** application.

The Intelligence Server interface comes in two parts:

INT386W.DLL A DLL, which contains the 6 functions that, constitute the Intelligence Server, interface. You load this DLL from your client program, making its functions available to the programming language you are using.

INT386W.OVL - The Intelligence Server Prolog overlay that implements the ***WIN-PROLOG*** side of the interface. This file handles the connection between the *INT386W.DLL* and the user-defined Prolog application code.

The limitations of an Intelligence Server application

The function calls to the *intelligence server* are monolithic; there is no way to interrupt a function call. So any calling application or thread will block waiting for the response. This means that it is important that you ensure that goals will terminate, otherwise you run the risk of the application or thread hanging, waiting on the return.

For example:

```
repeat, fail.
```

Would be a very bad goal!

This predicates that the Prolog code part of your application should be thoroughly tested prior to plugging it in to the *Intelligence Server*.

Chapter 2 Creating an Intelligence Server application

Before proceeding with the description of the *INT386W.DLL* and the Intelligence Server examples, this section shows how to build and deliver an application that uses the "Intelligence Server" toolkit. To follow the process through to the finish you should have already prepared the code for both the Prolog-end and the client, as described in later chapters.

A basic built "Intelligence Server" application consists of five components.

The client executable file (or equivalent collection of files).

The *INT386W.DLL* (supplied by LPA) containing the Intelligence Server functions.

The *INT386W.OVL* file contains the Prolog interface to the Intelligence Server.

Your code the Prolog code for your application.

The *INT386W.SYS* file the ***WIN-PROLOG*** run-time engine.

In addition there may be a further file for various language options, which interfaces between the *INT386W.DLL* and the front-end language, for example VB uses *VINT386W.DLL*.

Develop your Prolog code

A good first step is to ensure that your Prolog code for the server end is fully tested and debugged. As mentioned earlier, this can be done in complete isolation from the client language using the full programmer support features of the ***WIN-PROLOG*** development environment. A brief description of some *Intelligence Server* Prolog development tips is given in Chapter 9 Programming using the Intelligence Server.

Developing the client code

The next step is to build the client application that will call the *Intelligence Server*. Details are given in the following chapter of the available *INT386W.DLL* functions and how to call them. Additional information specific to each client language is given in chapters five to nine.

Delivering an Intelligence Server application

Having prepared all the components for the application, all that remains is the deployment. One way to deliver an *Intelligence Server* application is to copy the files *INT386W.SYS*, the appropriate *INT386W.DLL* for your client language, the *INT386W.OVL*, and the Prolog code for your application into the delivery directory along with the files associated with your application. Please note, in the present version of the *Intelligence Server*, you cannot re-name the above named files.

Chapter 3 The Intelligence Server Library

In the previous chapters we spoke about the philosophy of the Intelligence Server toolkit and how to build the ***WIN-PROLOG*** end of an *Intelligence Server* application. In this chapter we change the perspective to the front-end and look more closely at the *Intelligence Server* library and focus in on the details of the *Intelligence Server* functions.

As mentioned in the previous chapter, the communication between the client language and ***WIN-PROLOG*** is done via six functions provided in the *INT386W.DLL*. These functions are shown in the following table, and are described in more detail later on:

Int LoadProlog(SwitchesStr, Desired, Encoding, Tickle)	load a ***WIN-PROLOG*** application instance using the given switches, desired memory buffer size and text encoding and tickle flag, returning an identifier for the instance loaded.
UINT HaltProlog(Instance)	shut down a ***WIN-PROLOG*** application instance.
LPSTR InitGoal(Instance,Query)	initialise a Prolog goal for an instance.
LPSTR CallGoal(Instance)	call a previously initialised Prolog goal on an instance returning a solution, backtrack through the solutions to the goal on successive calls
BOOL ExitGoal(Instance)	exit from a goal on an instance once its solutions are no longer required
LPSTR TellGoal(InfoStr)	send information to a currently running goal requiring additional input

Table 1 - The Intelligence Server DLL functions

The Intelligence Server Functions

The *Intelligence Server* comprises a library of just six functions. Yet with these you can load and unload multiple ***WIN-PROLOG*** instances from memory, initiate nested queries, call and backtrack through alternatives, terminating them when no longer needed; you can even write Prolog code, which requires user input, and service this input from within your C/C++ (etc.) application. This way, your client code performs all the user interface functions, and ***WIN-PROLOG*** handles just the logic-based intelligence. Each of these functions is described in turn below.

LoadProlog(LPSTR Switches,UINT BufferSize, UINT Encoding, UINT Tickle)

Before you can perform any work with the *Intelligence Server* library, you must load ***WIN-PROLOG*** into memory. This function performs this operation, passing any command line switches as its first

argument, the size of the memory buffer required and the text encoding. The switches are specified in a NULL-terminated string, and the Boolean function returns TRUE or FALSE depending upon whether or not it succeeded. The BufferSize argument specifies the size in bytes of the interface buffer, if a size of 0 is specified then a default size of 64Kb is used. The Encoding argument can be either 0 or 1, 0 meaning that Prolog will handle all strings passed to it as ISO asciiz strings, and 1 meaning that Prolog will handle all strings passed to it as Unicode 16-bit wide-character strings. The Tickle argument can take a 0 or a 1 value, 0 meaning no tickle and 1 meaning tickle. The tickling ensures the prolog process does not fall down the windows priority list and thereby taking longer to wake up when a function is called to it.

In C, you could call it as follows:

```
UINT Id;

Id = LoadProlog("/H1024",0,0,0);

if (Id < 0)
{
   wsprintf(String,"LoadProlog Error %d!",Id) ;
   MessageBox(NULL,String,"CLIENT",MB_OK|MB_ICONHAND) ;
   return(FALSE) ;
} ;
```

This would try to load ***WIN-PROLOG*** with the "/H" switch (heap space) of 1024 (kilobytes). If you do not want to pass any switches, specify an empty string as the argument:

```
Id = LoadProlog("",0,0,0) ;
```

If this function returns a negative result, then the result is an error number and the *Intelligence Server* library has failed to load properly, and your application should report the error appropriately: none of the other functions will work.

HaltProlog(int Id)

To close a previously loaded instance of ***WIN-PROLOG*** , you should call this function, so in C:

```
HaltProlog(Id) ;
```

Once this has been done, none of the other *Intelligence Server* functions will work for that instance, apart, of course, from LoadProlog(Switches,BufferSize,Encoding,Tickle). You can load and halt ***WIN-PROLOG*** instances as often as you like during the running of an application.

Note that you should use this function to terminate a ***WIN-PROLOG*** , instance rather than try to execute *halt/0* as part of your *Intelligence Server* application code. The reason is that normally, when your queries are run by the *Intelligence Server* library, the CallGoal(Id) or TellGoal(Id,Input) functions (see below) wait for ***WIN-PROLOG*** to complete the query; if *halt/0* were executed, ***WIN-PROLOG*** would never return from the query, and the application would hang.

InitGoal(int Id, LPSTR Query)

In order to run a ***WIN-PROLOG*** query; you must first initialise it. This function takes an instance identifier and a NULL-terminated string, which should contain the valid Prolog text of a query. For example, from C, the call:

```
InitGoal( Id, "member( brown, [the,quick,brown,fox] ). ") ;
```

Will initialise, but not yet call, the Prolog goal:

```
member( brown, [the,quick,brown,fox] ).
```

This query will wait on *WIN-PROLOG* 's execution stack until it is called by CallGoal(Id) .

This function returns a string showing the result of the attempt to initialise the goal. If the attempt is successful the result will be the return code "G" followed by the 4-digit number that has been assigned to the goal. If the attempt is not successful the result will be the return code "E" followed by the 4-digit number that would have been assigned to the goal and the text of the error that occurred during the initialisation. Note that if an error occurs the goal stack is reset to 0000 and all pending queries on the stack for that instance are lost.

CallGoal(int Id)

Once a *WIN-PROLOG* query has been initialised, you can call it with this function: the result of the query is returned as a pointer to the output string, the type of this string depends on which *INT386W.DLL* you are using. A query, like the one shown above, either succeeds or fails, but generates no output; in such a case, the C call:

```
LPSTR Pointer ;
Pointer = CallGoal(Id) ;
```

will return with the "Pointer" variable pointing at a string such as:

```
"T 0012\n"
```

or

```
"F 0032\n"
```

The letter "T" indicates that the query succeeded, while "F" indicates that it failed. The 4-digit number is a query number, and indicates how deeply you have nested your queries. The first one that you initialise is assigned number 0000, the next one, 0001, and so on. You can use this number to keep track of which query is being responded to for this instance. When you exit from a query (see below), the number is automatically reduced by one as you re-enter the previously stacked query. The "\n" is C shorthand for "New Line", which in this case means "<ctrl-M><ctrl-J>".

There are two other return codes, "E" and "I". The former indicates that an error has occurred and has not been trapped by the *WIN-PROLOG* part of the application, and the latter that input has been requested by *WIN-PROLOG* . The full range of return codes are given in the following table:

Code	*Meaning*
G	*the goal was registered successfully*
T	*the called goal succeeded*
F	*the called goal failed*
E	*the called goal generated an error*
I	*the called goal is requesting information*

Table 2 - The Intelligence Server return codes

The "E" and "I" codes will be discussed in more detail further on.

ExitGoal(int Id)

When you have finished with a given ***WIN-PROLOG*** query, you can remove it from the execution stack with this function. The C call:

```
ExitGoal(Id) ;
```

will terminate the current query.

TellGoal(int Id, LPSTR Input)

Some ***WIN-PROLOG*** queries might require input from the user, and given that the rest of the user interface is implemented in C/C++ or whatever, it makes sense for this input to be supplied by the C interface. The *Intelligence Server* library includes a Prolog predicate, *input/2*, which can be used by Prolog code to request input. From within ***WIN-PROLOG*** , the goal:

```
input( ‘Please tell me your name‘, Name )
```

will cause the CallGoal() function to return with a string such as:

"I 0081\nPlease tell me your name"

The "I" indicates that input is required, and after the 4-digit query number and a new line, the text of the first argument to the *input/2* predicate is included. This can either be displayed to the user, while waiting for input, or can be used to index a relational database, name a file, or pretty much anything else. Once appropriate input has been generated, by whatever means, it should be returned to ***WIN-PROLOG*** using this function, for example as follows (in C):

```
LPSTR Pointer ;
Pointer = TellGoal(Id,"brian. ") ;
```

The *input/2* predicate will return the binding:

```
Name = ‘brian. ‘
```

and this Prolog string can be used as required, for example being read into a standard term:

```
read( Term ) <~ Name
```

or used as it is. It is entirely up to your ***WIN-PROLOG*** code how to handle the returned input string.

Just like CallGoal(Id), the TellGoal(Id,Input) function returns a pointer to another string. This might itself be a further request for input or it may be a solution originally requested by CallGoal(Id). Whenever the returned string begins with an "I", you should respond by calling TellGoal(Id,Input) with more input; if it begins with "T", "F" or "E", then the query has finished in success, failure or an error respectively.

Prolog Output: Returning Information

Apart from the *input/2* predicate described above, which returns a ***WIN-PROLOG*** specified prompt in addition to the "I" and 4-digit query number, no other information is returned from purely logical queries. Effectively, the "T" and "F" flags are simply equivalent to the "yes" and "no" responses of the interactive Prolog command line. In "normal" Prolog programs, the main way to provide information is to use output predicates. A program such as:

```
time :-
  time( _, _, _,Hour, Minutes, Seconds, _ ),
  write( 'The time is ' ),
  write( Hour-Minutes-Seconds ),
  nl.
```

could be used to display the current time. It does not "return" any data, and its success or failure is relatively unimportant: it is the output that matters. Using the *Intelligence Server* interface, this program could be called (from C) as follows:

```
LPSTR Pointer ;
InitGoal(Id,"time. ") ;
Pointer = CallGoal(Id) ;
ExitGoal(Id) ;
```

At this point, the "Pointer" variable will be left pointing at a NULL-terminated string of the form:

```
"T 0123\nThe time is 10 - 8 - 42\n"
```

Simply, all user output performed by each run of your query is copied to the output string returned by CallGoal(Id) or TellGoal(Id,Input). Apart from the initial 8 characters (T/F/E/I, <space>, ####, <ctrl-M>, <ctrl-J>) which are always presented, the rest of the output buffer depends entirely upon your ***WIN-PROLOG*** program.

Errors in Prolog Code

When CallGoal(Id) or TellGoal(Id,Input) return a string beginning with "E", it means an error has occurred which has resulted in your query being aborted. All stacked queries are also lost when this happens. The string returned will be of the form:

```
"E 0999\n<user output>\n<standard Prolog message>"
```

Any output your query managed to write prior to the error will be shown, followed by a standard Prolog error message, just like in the development environment. To avoid this happening, you should take steps to handle errors, either using *catch/2* to contain the effects of errors, or by writing an error handler (see below). Only if you are confident that your ***WIN-PROLOG*** code cannot generate errors, or if you are happy to pick up the pieces in your client application, should you ignore this advice!

After an error which has not been handled, you may start afresh with a new call to InitGoal(Id,Query): upon your first call to CallGoal(Id), the query number will have reset to "0000", indicating that this is the first query on the stack.

Handling Break, Error and Timer Events

Your ***WIN-PROLOG*** code can handle break, error and timer events in exactly the same way as in the development environment. All you need to do is define programs for one or more of the user-hook predicates, *?BREAK?/1*, *?ERROR?/2* or *?TIMER?/3*, and your code will be able to process the respective events as required.

By default, the *Intelligence Server* library transfers the above events to the *break_hook/1*, *error_hook/2* or *timer_hook/3*, predicates, and any events that you do not wish to handle yourself should be similarly directed.

Message Events and the ***WIN-PROLOG*** GUI

Pretty much the only thing that your ***WIN-PROLOG*** code cannot do when being driven as an Intelligence Server, is handle Windows messages. This is because the *Intelligence Server* interface itself uses these messages, and discards any that are not required. As a result, the handling of windows, dialogs, menus, and other GUI features is somewhat curtailed. While you can still create and display windows and dialogs in your Prolog code, perhaps for displaying debugging messages or some such, you will not be able to handle messages for them. Specifically, the following GUI predicates are not supported:

```
call_dialog/2
system_menu/3
wait/1
flag/1
```

Table 3 - predicates specifically not supported by the Intelligence Server

A particularly useful trick is to show the main console window by wshow(0,1). You can then write debugging information out to the console using for example, output(0),output(0),write(Hello world),nl,output(0). Certain other predicates, especially those to do with GUI-oriented user input, will also not work. You can safely use *msgbox/4* or *abtbox/3*, however, because messages for these are handled at a lower level.

Halting from ***WIN-PROLOG***

For reasons outlined earlier, your application code should not attempt to terminate ***WIN-PROLOG*** by calling the *halt/0*, *halt/1* or *exit/1* predicates. This is because the *Intelligence Server* interface always waits for your Prolog code to complete an action before returning to the client application. If you close down ***WIN-PROLOG*** with one of these predicates, the application will hang, waiting for the response from Prolog that will never come.

Chapter 4 The C/C++ Interface

In previous chapters we have looked at the *Intelligence Server* interface for ***WIN-PROLOG*** in a general manner, and described the six functions provided by the *Intelligence Server* library. In this chapter, we look at these functions in more detail from the perspective of the C programming language, and present a simple example application that can be compiled and run in conjunction with Microsoft C/C++ 1.5 or later.

Note for C++ users

To use the following functions in a C++ application, you need to declare them as C to avoid the C++ "name mangling". See CLIENT.EXE - A Worked C++ Example for an example using MFC where the Intelligence Server interface is encapsulated by an Int386w class.

The Intelligence Server functions

This section describes the types and return values of the *Intelligence Server* functions provided in the *INT386W.DLL*. Various programmatic considerations of calling the functions and using the interface are mentioned when they arise.

Int WINAPI LoadProlog(LPSTR Switches, UINT BufferSize, UINT Encoding, UINT Tickle) ;

This function is used to load a ***WIN-PROLOG*** instance into memory, passing it memory configuration switches as needed. The return value is a positive integer, that is the instance identifier if the function succeeded in loading the *Intelligence Server*, and a negative integer that is an error number if not.

```
UINT Id1, Id2, Id3,Id4;

Id1 = LoadProlog("",0,0,0) ;
Id2 = LoadProlog("/H512 /T1024",32000,0) ;
Id3 = LoadProlog((LPSTR)InitString,0,0) ;
Id4 = LoadProlog((LPSTR)InitString,0,0,1);
```

The first example passes no switches, and therefore loads ***WIN-PROLOG*** with its default setting, with the encoding as ISO and default buffer size of 64Kb. The second hardwires settings for the Heap and Text spaces of 512 and 1024 kilobytes respectively with a buffer size of 32000 bytes, and Unicode string encoding. While the third points to a string called "InitString"; which contains a NULL-terminated switch sequence, with a default buffer size of 64Kb and Unicode string encoding. The last example sets the Tickle flag on.

VOID WINAPI HaltProlog(int Id) ;

This function is used to terminate a ***WIN-PROLOG*** session, and should be used in place of *halt/0* in the Prolog application code. It takes an instance identifier, and returns no value. The correct form of call is:

```
HaltProlog( Id ) ;
```

LPSTR WINAPI InitGoal(int Id, LPSTR Goal) ;

This function is used to initialise a query goal for a Prolog instance, and returns no value. The query should be passed as a NULL-terminated string; examples of calls are:

```
InitGoal( Id1, "write('Hello World'). ") ;
InitGoal( Id2, (LPSTR)QueryString ) ;
```

The first example sets up the hard-wired the goal "write('Hello World')" in the instance Id1, while the second points to a string called "QueryString" which contains the NULL-terminated text of a query for instance Id2. Please note that all queries submitted must be terminated in ". " (<period><space>) in order to allow them to be read correctly in the *Intelligence Server* module.

LPSTR WINAPI CallGoal(int Id) ;

This function is used to execute a previously initialised query goal in a given instance. It returns a pointer to the result string; this string consists of an *Intelligence Server*-defined header of 8 bytes in the form:

```
XSYYYYNL
```

Where:

```
X       = T means success,  F means failure, E means an error occurred and I means a
request for input
S       = <space>
YYYY  = four-digit query number (0000 is first query)
NL            = new line (<ctrl-M><ctrl-J>)
```

Following this header is either output generated by the ***WIN-PROLOG*** application code; or an input prompt. Examples of calls are:

```
(LPSTR)ResultPointer = CallGoal(Id1);
lstrcpy((LPSTR)ResultBuffer,CallGoal(Id2)) ;
```

The first example assigns the returned pointer to the variable "ResultPointer", while the second copies the string to the buffer "ResultBuffer". Notice that all calls to CallGoal(Id) actually return the same string pointer, so if you want to keep results between calls, you will need to copy them as in the second example.

You can call this function repeatedly to force the current goal to backtrack through successive solutions.

LPSTR WINAPI TellGoal(int Id, LPSTR Input) ;

This function is used to provide input to a running query, before resuming the query. It returns a pointer to the result string (see above). This string may itself be a request for further input: it is important to understand that the TellGoal(Id,Input) function is the <u>only</u> valid response to an input request. Examples of calls are:

```
(LPSTR)ResultPointer = TellGoal(Id1,(LPSTR)InputString)) ;
lstrcpy((LPSTR)ResultBuffer,TellGoal(Id2,"Hello World")) ;
```

The first example passes input that has been stored in the string at "InputString", and records a pointer to the return string in the variable "ResultPointer", while the second passes the hardwired text "Hello World" and copies the return string to the buffer "ResultBuffer". Notice that all calls to TellGoal(Id,Input) actually return the same string pointer, so if you want to keep results between calls, you will need to copy them as in the second example.

VOID WINAPI ExitGoal(int Id) ;

This function is used to terminate a query for an instance, once no further solutions are required. It returns no value. The correct form of call is:

```
ExitGoal(Id) ;
```

After this function has been called, the next most recently used query becomes active once more.

CLIENT.EXE - A Worked C Example

In the rest of this chapter, we will put the theory into practice by describing a simple C application, which uses *INT386W* DLL to drive the ***WIN-PROLOG*** program, *CLIENT.PL*. This is itself simply a teletype-only version of the *MEALS.PL* example shipped with ***WIN-PROLOG*** . The *CLIENT.EXE* program is built from six files, which are described in the following sections.

CLIENT.BAT - The Make File

This file contains commands to control the Microsoft C compiler, which is used to build *CLIENT.EXE* from its constituent source and object files. You should note that certain paths are set explicitly at the start of this file. If you want to use the *CLIENT.BAT* program to create the *CLIENT.EXE* program you should change the following paths to suit your own system:

```
@echo off
set include=c:\sdk32\include
set lib=c:\sdk32\lib
set p=%path%
set path=c:\sdk32\bin;%path%
```

CLIENT.LNK - The Linker Directive File

This file contains commands to control the Microsoft *LINK.EXE* program, which is used to combine the compiled object files with appropriate run-time libraries:

INT386W.LIB - The INT386W DLL Function Declaration File

This file contains the *INT386W.DLL* exported function declarations to allow the linker to successfully link *CLIENT.EXE*.

CLIENT.H - The Header File

This file simply contains the definitions for a number of constants used in both *CLIENT.C* and *CLIENT.RC*:

```
#define     CLIENT_EDIT    20
#define     CLIENT_INIT    10
#define     CLIENT_CALL    11
#define     CLIENT_EXIT    12

#define     CLIENT_ANY     10
#define     CLIENT_GOOD    11
#define     CLIENT_CHEAP   12
#define     CLIENT_YUPPIE  13
#define     CLIENT_DIET    14
#define     CLIENT_GLUTTON     15
```

CLIENT.RC - The Resource Scripts

This file contains definitions of two dialogs, called "MealBox" and "UserBox". The former is used to drive the application, and the latter to obtain input from the user:

```
MealBox     DIALOG 50 50 245 120
STYLE DS_MODALFRAME | WS_POPUP | WS_SYSMENU
CAPTION     "CLIENT"
BEGIN
  EDITTEXT       CLIENT_EDIT   5  5 180 110,      ES_MULTILINE
  PUSHBUTTON "&Init Goal"  CLIENT_INIT 190  5  50  12,WS_TABSTOP
  PUSHBUTTON "&Call Goal"  CLIENT_CALL 190 20  50  12,  WS_TABSTOP
  PUSHBUTTON "&Exit Goal"  CLIENT_EXIT 190 35  50  12,  WS_TABSTOP
END

UserBox     DIALOG 30 40 115 52
STYLE DS_MODALFRAME | WS_POPUP
CAPTION     ""
BEGIN
  PUSHBUTTON "Any_Meal"    CLIENT_ANY          5  5 50 12,     WS_TABSTOP
  PUSHBUTTON "Good_Meal"  CLIENT_GOOD         5 20 50 12,     WS_TABSTOP
  PUSHBUTTON "Cheap_Meal" CLIENT_CHEAP        5 35 50 12,     WS_TABSTOP
  PUSHBUTTON "Yuppie_Meal" CLIENT_YUPPIE      60  5 50 12,    WS_TABSTOP
  PUSHBUTTON "Diet_Meal"    CLIENT_DIET          60 20 50 12,   WS_TABSTOP
  PUSHBUTTON "Glutton_Meal"      CLIENT_GLUTTON      60 35 50 12,
   WS_TABSTOP
END
```

CLIENT.C - The C Language Source Code

Finally we get to the main application source file, which loads *WIN-PROLOG* system into memory, followed by the *CLIENT.PL* example program, and then creates the "MealBox" dialog which runs for the duration of the *CLIENT* application. This dialog handles messages from three buttons, corresponding to the actions "Init Goal", "Call Goal" and "Exit Goal", and displays output from calls to *CLIENT*, which is running on the *Intelligence Server*. A secondary dialog, "UserBox", is created when necessary to elicit user input. First, the *CLIENT* and *Intelligence Server* functions are defined:

```
/*****************************************************************
 * Client Example for WIN-PROLOG - by Brian D Steel - 28 Oct 98      *
 * Copyright (c) 1998 - Logic Programming Associates Ltd             *
*****************************************************************/

#include     <windows.h>
#include     "client.h"

intWINAPI WinMain(HANDLE,HANDLE,LPSTR,int) ;
VOID   WINAPI ClientCallGoal() ;
BOOL   WINAPI MealProc(HWND,UINT,WPARAM,LPARAM) ;
BOOL   WINAPI UserProc(HWND,UINT,WPARAM,LPARAM) ;

UINT   WINAPI LoadProlog(LPSTR,UINT,UINT,UINT) ;
BOOL   WINAPI HaltProlog(UINT) ;
LPSTR WINAPI InitGoal(UINT,LPSTR) ;
LPSTR WINAPI CallGoal(UINT) ;
```

```
LPSTR WINAPI TellGoal(UINT,LPSTR) ;
BOOL  WINAPI ExitGoal(UINT) ;
```

Next, some global variables are declared:

```
intProlog ;
HANDLE    Instance ;
HWND      MealWindow ;
HWND      UserWindow ;
UCHAR     String[4096] ;
```

CLIENT.C - The WinMain Function

This function is the main backbone of any Windows application, and is responsible for initialising any global variables and setting up the application's main window. Here, it begins by saving the "instance" handle and two "procedure instances" which are needed later when defining dialogs:

```
// The main procedure loads Prolog and an example program and executes a dialog

int WINAPI WinMain(hInstance,hPrevInstance,lpCmdLine,nCmdShow)
HANDLE    hInstance ;
HANDLE    hPrevInstance ;
LPSTR     lpCmdLine ;
int       nCmdShow ;

{
   Instance = hInstance ;
```

Next, an attempt is made to load ***WIN-PROLOG*** by calling the "LoadProlog" function in the *Intelligence Server*. If the load is unsuccessful (for example, because Prolog cannot be found), a message box is shown and the "WinMain" function exits with the value "FALSE", terminating the *CLIENT* application:

```
Prolog = LoadProlog("",0,0,0) ;
if  (Prolog < 0)
{
   wsprintf(String,"LoadProlog Error %d!",Prolog) ;
   MessageBox(NULL,String,"CLIENT",MB_OK|MB_ICONHAND) ;
   return(FALSE) ;
} ;
```

Provided ***WIN-PROLOG*** loaded correctly, a call is made via the *Intelligence Server* to load the example file, *CLIENT.PL*, which should be in the Prolog's home directory:

```
InitGoal(Prolog,"consult(client). ") ;
CallGoal(Prolog) ;
ExitGoal(Prolog) ;
```

All that remains is to create the "MealBox" dialog, and when it is complete, to close down the *Intelligence Server* by calling the "HaltProlog" function, before returning "TRUE" and terminating the *CLIENT* application:

```
DialogBox(Instance,"MealBox",NULL,MealProc) ;
HaltProlog(Prolog) ;
```

```
return(TRUE) ;
}
```

CLIENT.C - The ClientCallGoal Function

This function wraps up two *Intelligence Server* functions, namely "CallGoal" and "TellGoal", providing user input from the "UserBox" dialog whenever one of these functions returns output beginning with the letter I (for "input"):

```
// call the given goal, and loop round performing input as required
VOID WINAPI ClientCallGoal(Prolog)
{
   lstrcpy(String,CallGoal(Prolog)) ;

   while (String[0] == 'I')
   {
        DialogBox(Instance,"UserBox",MealWindow,UserProc) ;
        lstrcpy(String,TellGoal(Prolog,(LPSTR)String)) ;
   } ;
```

Once either "CallGoal" or "TellGoal" has returned output not beginning with the letter I, a check is made to see if there is any output at all; if not, the return string is set to the message, "NO PENDING QUERIES":

```
   if (!String[0])
   {
        lstrcpy(String,"NO PENDING QUERIES") ;
   } ;

   return ;
}
```

CLIENT.C - The MealProc Function

This function handles messages for the "MealBox" dialog:

```
BOOL  WINAPI MealProc(Window,Message,wParam,lParam)
HWND       Window ;
UINT       Message ;
WPARAM     wParam ;
LPARAM     lParam ;
{
   switch(Message)
   {
```

The first message, "WM_INITDIALOG", simply stores the dialog window handle in a global variable, "MealWindow":

```
        caseWM_INITDIALOG :
             MealWindow = Window ;
             break ;
```

The second message, "WM_COMMAND", signifies that one of the buttons, named "CLIENT_INIT", "CLIENT_CALL" or "CLIENT_EXIT", has been pressed. The first of these calls the *Intelligence Server*

functions, "InitGoal" and "CallGoal"; the second just calls "CallGoal", while the third calls "ExitGoal" and "CallGoal":

```
        caseWM_COMMAND :
            switch      (wParam)
            {
                  case  CLIENT_INIT :
                        InitGoal(Prolog,"menu. ") ;
                        ClientCallGoal(Prolog) ;
                  break ;

                  case  CLIENT_CALL :
                        ClientCallGoal(Prolog) ;
                  break ;

                  case  CLIENT_EXIT :
                        ExitGoal(Prolog) ;
                        ClientCallGoal(Prolog) ;
                  break ;
            } ;
```

After any of the buttons, the "CLIENT_EDIT" dialog window is set to contain the text stored in the global buffer, "String":

```
            SendDlgItemMessage(Window,
                              CLIENT_EDIT,
                              WM_SETTEXT,
                              (WPARAM) NULL,
                              (LPARAM) String) ;

            return(TRUE) ;
        break ;
```

The third message, "WM_CLOSE", is used to end the dialog successfully, which will cause the *CLIENT* application to terminate:

```
        caseWM_CLOSE :
            EndDialog(Window,TRUE) ;
            return(TRUE) ;
            break ;
```

The fourth message, "WM_DESTROY", simply resets the global "MealWindow" variable:

```
        caseWM_DESTROY :
            MealWindow = NULL ;
            break ;
```

Finally, for those message which do not "return" explicitly, and for all messages which are not otherwise handled, the dialog function returns the value "FALSE", which tells Windows to perform appropriate default processing:

```
        } ;
        return(FALSE) ;
    }
```

CLIENT.C - The UserProc Function

This function handles messages for the "UserBox" dialog:

```
BOOL WINAPI UserProc(Window,Message,wParam,lParam)
HWND        Window ;
UINT        Message ;
WPARAM      wParam ;
LPARAM      lParam ;
{
   switch    (Message)
   {
```

The first message, "WM_INITDIALOG", stores the dialog window handle in a global variable, "UserWindow", and then sets the caption of the dialog to the text stored in the global buffer, "String", omitting the *Intelligence Server* header in the first eight bytes:

```
caseWM_INITDIALOG :
    UserWindow = Window ;
    SendMessage( Window,
                 WM_SETTEXT,
                 (WPARAM) NULL,
                 (LPARAM) String+8 ) ;
    break ;
```

The second message, "WM_COMMAND", signifies that one of the buttons has been pressed: a check is made for each one in turn, and the text of the appropriate Prolog term is copied to the global buffer, "String", before terminating the dialog:

```
caseWM_COMMAND :
    switch      (wParam)
    {
    case  CLIENT_ANY :
          lstrcpy(String,"any_meal. ") ;
          break ;

    case  CLIENT_GOOD :
          lstrcpy(String,"good_meal. ") ;
          break ;

    case  CLIENT_CHEAP :
          lstrcpy(String,"cheap_meal. ") ;
          break ;

    case  CLIENT_YUPPIE :
          lstrcpy(String,"yuppie_meal. ") ;
          break ;

    case  CLIENT_DIET :
          lstrcpy(String,"diet_meal. ") ;
          break ;

    case  CLIENT_GLUTTON :
lstrcpy(String,"glutton_meal. ") ;
```

```
        break ;
    } ;

    EndDialog(Window,TRUE) ;
    return(TRUE) ;
    break ;
```

The third message, "WM_CLOSE", is explicitly discarded by returning a value of "TRUE", preventing the user from closing this dialog with <alt-F4>:

```
    caseWM_CLOSE :
        return(TRUE) ;
        break ;
```

The fourth message, "WM_DESTROY", simply resets the global "UserWindow" variable:

```
    caseWM_DESTROY :
        UserWindow = NULL ;
        break ;
```

Finally, for those messages which do not "return" explicitly, and for all messages which are not otherwise handled, the dialog function returns the value "FALSE", which tells Windows to perform appropriate default processing:

```
    } ;
    return(FALSE) ;
}
```

Building the CLIENT.EXE Application

To build *CLIENT.EXE* from the supplied source files, you will need the 32-bit compiler from Microsoft's Visual C 1.5 or later compiler, together with the appropriate linker and make utilities. All being well, the DOS command:

```
C> CLIENT.BAT
```

will cause the source files to be compiled and the object and resource files to be linked into the application. If you do not have this compiler, but use another version of C, modify the source files as necessary and build the system accordingly. If you have no C compiler, but merely wish to try out *CLIENT.EXE*, a ready-built version has been included with the source files.

CLIENT.EXE - A Worked C++ Example

Having described the principals of how to use the Intelligence Server from C, we now present a more modern example of using C++ and specifically Microsoft Foundation Classes™ (MFC). Please bear in mind that Wizards in the Visual C++ 6 environment will automatically generate a lot of this code.

Client.h - The Client Header file

```
// Client.h : main header file for the CLIENT application
//

#if !
defined(AFX_CLIENT_H__70F08A79_B308_4CA1_96C6_8CB3E7E03699__INCLUDED_
```

```
)
#define AFX_CLIENT_H__70F08A79_B308_4CA1_96C6_8CB3E7E03699__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"              // main symbols

/////////////////////////////////////////////////////////////////////////////
// CClientApp:
// See Client.cpp for the implementation of this class
//

class CClientApp : public CwinApp
{
public:
    CClientApp();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CClientApp)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Implementation

    //{{AFX_MSG(CClientApp)
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !
defined(AFX_CLIENT_H__70F08A79_B308_4CA1_96C6_8CB3E7E03699__INCLUDED_
)
```

Client.cpp - The Main Application File

```
// Client.cpp : Defines the class behaviors for the application.
//
```

```
#include "stdafx.h"
#include "Client.h"
#include "ClientDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CClientApp

BEGIN_MESSAGE_MAP(CClientApp, CWinApp)
//{{AFX_MSG_MAP(CClientApp)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code
    //}}AFX_MSG
    ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CClientApp construction

CClientApp::CClientApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CClientApp object

CClientApp theApp;

/////////////////////////////////////////////////////////////////////////////
// CClientApp initialisation

BOOL CClientApp::InitInstance()
{
    // Standard initialisation
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();                 // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();  // Call this when linking to MFC statically
#endif
```

Create the main dialog window, and call it modally.

```
    CClientDlg dlg;
    m_pMainWnd = &dlg;
    int nResponse = dlg.DoModal();

    // Since the dialog has been closed, return FALSE so that we exit the
    //  application, rather than start the application's message pump.
    return FALSE;
}
```

Client.rc - The Resource File

This file contains definitions for the dialogs used by the application.

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_CLIENT_DIALOG DIALOGEX 0, 0, 320, 148
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "CLIENT"
FONT 8, "MS Sans Serif"
BEGIN
   PUSHBUTTON     "&Init Goal",IDC_INIT_GOAL,263,7,50,14
   PUSHBUTTON     "&Call Goal",IDC_CALL_GOAL,263,25,50,14
   PUSHBUTTON     "&Exit Goal",IDC_EXIT_GOAL,263,43,50,14
   EDITTEXT   IDC_EDIT1,7,7,251,134,ES_MULTILINE | ES_AUTOVSCROLL |
              ES_AUTOHSCROLL
END
```

```
#ifndef _MAC
/////////////////////////////////////////////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x4L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904B0"
        BEGIN
            VALUE "CompanyName", "\0"
            VALUE "FileDescription", "Client MFC Application\0"
            VALUE "FileVersion", "1, 0, 0, 1\0"
            VALUE "InternalName", "Client\0"
            VALUE "LegalCopyright", "Copyright (C) 2001\0"
            VALUE "LegalTrademarks", "\0"
            VALUE "OriginalFilename", "Client.EXE\0"
            VALUE "ProductName", "Client Application\0"
            VALUE "ProductVersion", "1, 0, 0, 1\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END

#endif    // !_MAC


/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
```

```
    IDD_CLIENT_DIALOG, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 313
        TOPMARGIN, 7
        BOTTOMMARGIN, 141
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////
// English (U.K.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENG)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_UK
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
    "#define _AFX_NO_OLE_RESOURCES\r\n"
    "#define _AFX_NO_TRACKER_RESOURCES\r\n"
    "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
    "\r\n"
    "#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)\r\n"
    "#ifdef _WIN32\r\n"
    "LANGUAGE 9, 1\r\n"
    "#pragma code_page(1252)\r\n"
    "#endif //_WIN32\r\n"
    "#include ""res\\Client.rc2""  // non-Microsoft Visual C++ edited resources\r\n"
```

```
   "#include ""afxres.rc""         // Standard components\r\n"
   "#endif\r\n"
   "\0"
END

#endif    // APSTUDIO_INVOKED


////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\Client.ico"

////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_USERBOX DIALOG DISCARDABLE  0, 0, 120, 63
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Select a Meal"
FONT 8, "MS Sans Serif"
BEGIN
   PUSHBUTTON      "Any Meal",IDC_ANYMEAL,7,7,50,14
   PUSHBUTTON      "Good Meal",IDC_GOODMEAL,7,25,50,14
   PUSHBUTTON      "Cheap Meal",IDC_CHEAPMEAL,7,43,50,14
   PUSHBUTTON      "Yuppie Meal",IDC_YUPPIEMEAL,63,7,50,14
   PUSHBUTTON      "Diet Meal",IDC_DIETMEAL,63,25,50,14
   PUSHBUTTON      "Glutton Meal",IDC_GLUTTONMEAL,63,43,50,14
END


////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
   IDD_USERBOX, DIALOG
   BEGIN
      LEFTMARGIN, 7
      RIGHTMARGIN, 113
      TOPMARGIN, 7
      BOTTOMMARGIN, 56
   END
END
#endif    // APSTUDIO_INVOKED
```

```
#endif    // English (U.K.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCE
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE 9, 1
#pragma code_page(1252)
#endif //_WIN32
#include "res\Client.rc2"  // non-Microsoft Visual C++ edited resources
#include "afxres.rc"         // Standard components
#endif

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

ClientDlg.h - The header file for the main dialog

This file contains class definitions for the main dialog.

```
// ClientDlg.h : header file
//
#include "Int386w.hpp"
#include "mealdlg.h"

#if !
defined(AFX_CLIENTDLG_H__E72C95F7_C64C_413C_BA83_E32D36397CBB__INCLUD
ED_)
#define
AFX_CLIENTDLG_H__E72C95F7_C64C_413C_BA83_E32D36397CBB__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CClientDlg dialog

class CClientDlg : public CDialog
{
// Construction
public:
```

```
    void ClientCallGoal();
    CClientDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CClientDlg)
    enum { IDD = IDD_CLIENT_DIALOG };
    CEdit      m_edit;
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CClientDlg)
    public:
    virtual int DoModal();
    virtual CScrollBar* GetScrollBarCtrl(int nBar) const;
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    HICON m_hIcon;
```

This declares an Intelligence Server instance for this class.

```
    Int386w m_prolog;

    // Generated message map functions
    //{{AFX_MSG(CClientDlg)
    virtual BOOL OnInitDialog();
    afx_msg void OnPaint();
    afx_msg HCURSOR OnQueryDragIcon();
    afx_msg void OnCallGoal();
    afx_msg void OnInitGoal();
    afx_msg void OnExitGoal();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !
defined(AFX_CLIENTDLG_H__E72C95F7_C64C_413C_BA83_E32D36397CBB__INCLUD
ED_)
```

ClientDlg.cpp - The implementation file for the main dialog

```
// ClientDlg.cpp : implementation file
//

#include "stdafx.h"
```

```
#include "Client.h"
#include "ClientDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CClientDlg dialog

CClientDlg::CClientDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CClientDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CClientDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CClientDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CClientDlg)
    DDX_Control(pDX, IDC_EDIT1, m_edit);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CClientDlg, CDialog)
    //{{AFX_MSG_MAP(CClientDlg)
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_CALL_GOAL, OnCallGoal)
    ON_BN_CLICKED(IDC_INIT_GOAL, OnInitGoal)
    ON_BN_CLICKED(IDC_EXIT_GOAL, OnExitGoal)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CClientDlg message handlers

BOOL CClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);                 // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    return TRUE;  // return TRUE  unless you set the focus to a control
```

```
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CClientDlg::OnPaint()
{
   if (IsIconic())
   {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
   }
   else
   {
        CDialog::OnPaint();
   }
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CClientDlg::OnQueryDragIcon()
{
   return (HCURSOR) m_hIcon;
}
```

The call goal button handler, when the call goal button is clicked ClientCallGoal() is called to call again the current goal and backtrack through solutions. The returned text is copied to the edit control.

```
void CClientDlg::OnCallGoal()
{
   ClientCallGoal();
   m_edit.SetWindowText( m_prolog.m_text );
}
```

The init goal button handler, when the init goal button is clicked the menu. prolog goal is initialised by calling m_prolog.initGoal and the first call is made through ClientCallGoal() and the returned text is copied to the edit control.

```
void CClientDlg::OnInitGoal()
{
   m_prolog.initGoal( "menu. " );
   ClientCallGoal();
   m_edit.SetWindowText( m_prolog.m_text );
}
```

The exit goal button handler, when the exit goal button is clicked the current goal is terminated by calling m_prolog.exitGoal() and the previously stacked goal is called again through ClientCallGoal(). The returned text is copied to the edit control.

```
void CClientDlg::OnExitGoal()
{
   m_prolog.exitGoal();
   ClientCallGoal();
   m_edit.SetWindowText( m_prolog.m_text );
}

int CClientDlg::DoModal()
{
  m_prolog.callOneGoal( "consult(prolog(client)). " );
   return CDialog::DoModal();
}

CScrollBar* CClientDlg::GetScrollBarCtrl(int nBar) const
{
   return CDialog::GetScrollBarCtrl(nBar);
}
```

ClientCallGoal() is where all the work is done to call prolog.

```
void CClientDlg::ClientCallGoal()
{
```

A call is made to the current goal through m_prolog.callGoal(), copying the result to a temporary buffer.

```
   lstrcpy( m_prolog.m_text, m_prolog.callGoal() );
```

If the previous call requests more information, by returning 'I' as the result character then a loop is entered.

```
   while( m_prolog.m_text[0] == 'I' )
   {
     CMealDlg dlg;
```

The returned text is copied to the edit control.

```
     dlg.SetWindowText( m_prolog.m_text);
```

And the "Choose meal" dialog is called.

```
int nResponse = dlg.DoModal();
```

On the result from the dialog the appropriate meal goal is chosen to return to prolog.

Notice that the default is any_meal.

```
switch( nResponse )
{
        default:
        case IDC_ANYMEAL:
                lstrcpy( m_prolog.m_text, "any_meal. " ) ;
                break;
        case IDC_GOODMEAL:
                lstrcpy( m_prolog.m_text, "good_meal. " ) ;
                break;
        case IDC_CHEAPMEAL:
                lstrcpy( m_prolog.m_text, "cheap_meal. " ) ;
                break;
        case IDC_YUPPIEMEAL:
                lstrcpy( m_prolog.m_text, "yuppie_meal. " ) ;
                break;
        case IDC_DIETMEAL:
                lstrcpy( m_prolog.m_text, "diet_meal. " ) ;
                break;
        case IDC_GLUTTONMEAL:
                lstrcpy( m_prolog.m_text, "glutton_meal. " ) ;
                break;
}
```

Prolog is then told that goal.

```
  lstrcpy( m_prolog.m_text, m_prolog.tellGoal(m_prolog.m_text) );
}
```

If there is no return from prolog at that point, the text is set to indicate there are no remaining queries left on the call stack.

```
  if( (m_prolog.m_text[0]) == 0 )
    lstrcpy( m_prolog.m_text, "No Pending Queries" );
}
```

Int386w.hpp - The header file for the Intelligence Server Class

This file declares the class definitions for the Intelligence Server Class.

```
// Int386w.hpp: interface for the Int386wclass.
//
//////////////////////////////////////////////////////////////////////
```

```
#if !
defined(AFX_INT386W_HPP__8EC8CA7C_860D_4C62_89D3_C71E326DA12B__INCLUD
ED_)
#define
AFX_INT386W_HPP__8EC8CA7C_860D_4C62_89D3_C71E326DA12B__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

This is important, the functions in the INT386W.DLL are declared as C function to avoid name mangling.

```
extern "C"
{
   UINT  WINAPI LoadProlog( LPSTR, UINT, UINT, UINT );
   BOOL  WINAPI HaltProlog( UINT );

   LPSTR WINAPI InitGoal( UINT, LPSTR );
   LPSTR WINAPI CallGoal( UINT );
   BOOL  WINAPI ExitGoal( UINT );
   LPSTR WINAPI TellGoal( UINT, LPSTR );

}
```

Declaration of error constants.

```
const  int isERR_ENCODE    = -1;
const  int isERR_TICKLE    = -2;
const  int isERR_CMUTEX    = -3;
const  int isERR_WMUTEX    = -4;
const  int isERR_LOCATE    = -5;
const  int isERR_CREATE    = -6;
const  int isERR_MAPFIL    = -7;
const  int  isERR_MODULE   = -8;
const  int isERR_PROLOG    = -9;
const  int isERR_ACTIVE    = -10;
const  int isERR_WINDOW    = -11;
```

The class, notice that the prolog id is not required when calling functions here. Each new instance of Int386w represents an instance of the Intelligence Server engine.

The constructor defines the defaults for loading prolog of command line = , bufferSize = 0, and encryption = 0 (iso character set) and tickle=0 (no tickling please!).

```
class Int386w
{
 public:
   char m_text[65535];
   LPSTR callOneGoal( LPSTR Goal );
   int loadError;
   int id;
```

```
    Int386w(LPSTR Command="", UINT bufferSize = 0, UINT encryption = 0, UINT
tickle = 0 );
    virtual ~Int386w();

    LPSTR initGoal(LPSTR Goal);
    LPSTR callGoal();
    BOOL  exitGoal();
    LPSTR tellGoal( LPSTR Goal );
};

#endif // !
defined(AFX_INT386W_HPP__8EC8CA7C_860D_4C62_89D3_C71E326DA12B__INCLUD
ED_)
```

Int386w.cpp - The implementation file for the Intelligence Server Class

The implementation of the Intelligence Server class.

```
// Int386w.cpp: implementation of the Int386w class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Client.h"
#include "Int386w.hpp"


#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////
```

The constructor for the class calls the loadProlog member function to initialise the Intelligence Server.

```
Int386w::Int386w(LPSTR Command, UINT bufferSize, UINT encryption, UINT tickle )
{
    id = 0;
    loadError = 0;
    int temp = LoadProlog( Command, bufferSize, encryption, tickle );

    if( temp > 0 )
    {
        id = temp;
        loadError = 0;
```

```
    } else
    {
        id = 0;
        loadError = temp;
    }
}
```

The destructor calls haltProlog() if the id has been set, then the id is set to 0.

```
Int386w::~Int386w()
{
    if( id ) HaltProlog();
    id = 0;
}
```

This sets up a goal, for the current instance

```
// Setup a goal for this instance
LPSTR Int386w::initGoal( LPSTR Goal )
{
    return InitGoal( id, Goal );
}
```

Call a previously set up goal returning the string result.

```
// Call a previously setup goal for this instances
LPSTR Int386w::callGoal()
{
    return CallGoal( id );
}

// Exit a goal, bringing the previous goal to the top of the goal stack
BOOL Int386w::exitGoal()
{
    return ExitGoal(id);
}

// Tell a goal
LPSTR Int386w::tellGoal( LPSTR Goal )
{
    return TellGoal( id, Goal );
}

// Call a goal once only
LPSTR Int386w::callOneGoal(LPSTR Goal)
{
  lstrcpy( m_text, initGoal(Goal) );
  lstrcpy( m_text, callGoal() );
  ExitGoal( id );
```

```
  return( m_text );
}
```

MealDlg.h - The header file for the meal selection dialog.

```
#if !
defined(AFX_MEALDLG_H__09B39837_49B4_43C2_A76C_79D2345300CE__INCLUDE
D_)
#define
AFX_MEALDLG_H__09B39837_49B4_43C2_A76C_79D2345300CE__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// MealDlg.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CMealDlg dialog

class CMealDlg : public CDialog
{
// Construction
public:
    virtual int DoModal();
    CMealDlg(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CMealDlg)
    enum { IDD = IDD_USERBOX };
        // NOTE: the ClassWizard will add data members here
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMealDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CMealDlg)
    afx_msg void OnAnymeal();
    afx_msg void OnCheapmeal();
    afx_msg void OnDietmeal();
    afx_msg void OnGluttonmeal();
    afx_msg void OnGoodmeal();
    afx_msg void OnYuppiemeal();
```

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !
defined(AFX_MEALDLG_H__09B39837_49B4_43C2_A76C_79D2345300CE__INCLUDE
D_)
```

MealDlg.cpp - The implementation file for the meal selection dialog.

```
// MealDlg.cpp : implementation file
//

#include "stdafx.h"
#include "Client.h"
#include "MealDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMealDlg dialog


CMealDlg::CMealDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CMealDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CMealDlg)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}


void CMealDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMealDlg)
        // NOTE: the ClassWizard will add DDX and DDV calls here
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CMealDlg, CDialog)
    //{{AFX_MSG_MAP(CMealDlg)
    ON_BN_CLICKED(IDC_ANYMEAL, OnAnymeal)
```

```
    ON_BN_CLICKED(IDC_CHEAPMEAL, OnCheapmeal)
    ON_BN_CLICKED(IDC_DIETMEAL, OnDietmeal)
    ON_BN_CLICKED(IDC_GLUTTONMEAL, OnGluttonmeal)
    ON_BN_CLICKED(IDC_GOODMEAL, OnGoodmeal)
    ON_BN_CLICKED(IDC_YUPPIEMEAL, OnYuppiemeal)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMealDlg message handlers

void CMealDlg::OnAnymeal()
{
    CDialog::EndDialog(IDC_ANYMEAL);
}

void CMealDlg::OnCheapmeal()
{
    CDialog::EndDialog(IDC_CHEAPMEAL);
}

void CMealDlg::OnDietmeal()
{
    CDialog::EndDialog(IDC_DIETMEAL);
}

void CMealDlg::OnGluttonmeal()
{
    CDialog::EndDialog(IDC_GLUTTONMEAL);
}

void CMealDlg::OnGoodmeal()
{
    CDialog::EndDialog(IDC_GOODMEAL);
}

void CMealDlg::OnYuppiemeal()
{
    CDialog::EndDialog(IDC_YUPPIEMEAL);
}

int CMealDlg::DoModal()
{
  return CDialog::DoModal();
}
```

Resource.h - A header file containing resource constants.

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Client.rc
//
```

```
#define IDD_CLIENT_DIALOG 102
#define IDR_MAINFRAME           128
#define IDD_USERBOX             129
#define IDC_INIT_GOAL           1000
#define IDC_CALL_GOAL           1001
#define IDC_EXIT_GOAL           1002
#define IDC_EDIT1               1003
#define IDC_ANYMEAL             1004
#define IDC_GOODMEAL            1005
#define IDC_CHEAPMEAL           1006
#define IDC_YUPPIEMEAL          1007
#define IDC_DIETMEAL            1008
#define IDC_GLUTTONMEAL     1009

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE    130
#define _APS_NEXT_COMMAND_VALUE     32771
#define _APS_NEXT_CONTROL_VALUE     1005
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

StdAfx.cpp and StdAfx.h - files for including system includes

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !
defined(AFX_STDAFX_H__D183AC51_3813_4076_A397_8808C0E87D1F__INCLUDED_
)
#define AFX_STDAFX_H__D183AC51_3813_4076_A397_8808C0E87D1F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN            // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdtctl.h>           // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>                 // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT



//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
```

previous line.

```
#endif // !
defined(AFX_STDAFX_H__D183AC51_3813_4076_A397_8808C0E87D1F__INCLUDED_
)

// stdafx.cpp : source file that includes just the standard includes
//   Client.pch will be the pre-compiled header
//   stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

Chapter 5 The Delphi Interface

In the previous chapter, we looked at the *Intelligence Server* interface from a C point of view. The descriptions of the *Intelligence Server* functions hold true for the Delphi interface as well. All that remains to be done in this chapter is to present a simple Delphi example application.

CLIENT.EXE - A Worked Example

In this chapter, we will describe a simple Delphi application, which uses *INT386W.DLL* to drive the ***WIN-PROLOG*** program, *CLIENT.PL*. This is itself simply a teletype-only version of the *MEALS.PL* example shipped with ***WIN-PROLOG*** . The *CLIENT.EXE* program is built from six files, which are described in the following sections.

CLIENT.DPR - The Project File

This file is the project file, which is used to build *CLIENT.EXE* from its constituent source files; this can be loaded into the Delphi environment in the usual way:

```
program client;
uses
  Forms,
  client2 in 'client2.pas' {Choice},
  client1 in 'client1.pas' {Form1},
  IntelligenceServer in 'IntelligenceServer.PAS';

{$R *.RES}

begin
Application.Initialize;
  Application.CreateForm(TMain, Main);
  Application.CreateForm(TChoice, Choice);
  Application.Run;
end.
```

INTELLIGENCESERVER.PAS - The Intelligence Server DLL Interface Declarations

This file contains the declaration unit for the *INT386W.DLL*, and should be included in your project.

This unit is included in the "uses" section of the *CLIENT1.PAS* file.

```
{ LPA-Delphi Logic Server
  PROGRAM: INTELLIGENCESERVER.PAS
  PURPOSE: Interface unit to the intelligence server DLL
  CREATED: May 1996
  MODIFIED for unicode: February 2001
  MODIFIED for changes to LoadProlog : March 2003
  AUTHOR:  Alan Westwood

  Copyright (c) 2003 LPA Ltd.
}
```

```
// Defining LPAUNICODE before compilation allows the use of wide strings
// but in that case String constants will need to be type cast with
// PChar( 'fred' ) or PWideChar( 'fred' ) .
// use either LoadProlog encoding 0 or encoding 1 ( eg. LoadProlog( "", 0, 1, 0 )
// or LoadProlog( "", 0, 0, 0 ) )
// Not defining LPAUNICODE will mean that the interface will work in it's old
// Ansi string mode.
// use LoadProlog encoding 0, ( eg. LoadProlog("",0,0,0) ) otherwise PChar will
// be interpreted as PWideChar
// within the interface.

//{$DEFINE LPAUNICODE}

unit IntelligenceServer;

interface

  const isERR_ENCODE = -1;
  const isERR_TICKLE = -2;
  const isERR_CMUTEX = -3;
  const isERR_WMUTEX = -4;
  const isERR_LOCATE = -5;
  const isERR_CREATE = -6;
  const isERR_MAPFIL = -7;
  const isERR_MODULE = -8;
  const isERR_PROLOG = -9;
  const isERR_ACTIVE = -10;
  const isERR_WINDOW = -11;

{ prototypes }
  function  LoadProlog( CommandLine : PChar; Desire, Encode, Tickle : Integer   ) :
Integer; stdcall;
  procedure HaltProlog( PrologID : Integer ) ; stdcall;
  procedure ExitGoal( PrologID : Integer )  ; stdcall;

{$IFDEF LPAUNICODE }
  function  InitGoal( PrologID : Integer; Input : PChar )     : PChar ; stdcall; overload;
  function  InitGoal( PrologID : Integer; Input : PWideChar ) : PWideChar ; stdcall;
overload;
  function  TellGoal( PrologID : Integer; Input : PChar )     : PChar ; stdcall; overload;
  function  TellGoal( PrologID : Integer; Input : PWideChar ) : PWideChar ; stdcall;
overload;
  function  CallGoal( PrologID : Integer ) : Pointer ; stdcall;
{$ELSE}
  function  InitGoal( PrologID : Integer; Input : PChar )   : PChar ; stdcall;
  function  TellGoal( PrologID : Integer; Input : PChar )   : PChar ; stdcall;
  function  CallGoal( PrologID : Integer )                  : PChar ; stdcall;
{$ENDIF}

implementation
  function  LoadProlog( CommandLine : PChar; Desire, Encode, Tickle : Integer ) :
Integer; stdcall;  external 'INT386W.DLL';
  procedure HaltProlog( PrologID : Integer ); stdcall;  external 'INT386W.DLL';
```

```
  procedure ExitGoal( PrologID : Integer );  stdcall;  external 'INT386W.DLL';

{$IFDEF LPAUNICODE }
  function  InitGoal( PrologID : Integer; Input : PChar )     : PChar; stdcall;  external
'INT386W.DLL';
  function  InitGoal( PrologID : Integer; Input : PWideChar ) : PWideChar; stdcall;
external 'INT386W.DLL';
  function  TellGoal( PrologID : Integer; Input : PChar )     : PChar; stdcall; external
'INT386W.DLL';
  function  TellGoal( PrologID : Integer; Input : PWideChar ) : PWideChar; stdcall;
external 'INT386W.DLL';
  function  CallGoal( PrologID : Integer ) : Pointer; stdcall;  external 'INT386W.DLL';
{$ELSE}
  function  InitGoal( PrologID : Integer; Input : PChar ) : PChar; stdcall;  external
'INT386W.DLL';
  function  CallGoal( PrologID : Integer ) : PChar; stdcall;  external 'INT386W.DLL';
  function  TellGoal( PrologID : Integer; Input : PChar ) : PChar; stdcall; external
'INT386W.DLL';
{$ENDIF}

end.
```

CLIENT1.DFM - Form Description

This file contains the Delphi generated description of the main form.

CLIENT2.DFM - Form Description

This file contains the Delphi generated form description for the input form.

CLIENT1.PAS - The Main Form

We now turn to the main application source file, which creates the form "Main". When this form is created the procedure "FormCreate" is executed. This procedure loads the ***WIN-PROLOG*** system into memory, followed by the *CLIENT.PL* example program.

The form has three buttons, "Init Goal", "Call Goal" and "Exit Goal". Hitting the "Init Goal" button causes the application to initialise and call the Prolog goal: "menu. ". Hitting the "Call Goal" button causes the application to call the previously initialised Prolog goal. Hitting the "Exit Goal" button causes the current Prolog goal to be exited, and any previous goal to be called once more. Closing the application will tell Prolog to halt.

If, in the process of executing a goal Prolog requires more `data` a second form is called to get that data from the user.

```
unit client1;

interface
```

The following "uses" section includes the server interface unit, and

The input form, client2:

```
uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  StdCtrls,
IntelligenceServer, client2;
```

The following is the type declaration for the main form:

```
type

  TMain = class(TForm)
  OutputMemo: TMemo;
  InitGoalButton: TButton;
  CallGoalButton: TButton;
  ExitGoalButton: TButton;
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
  procedure InitGoalButtonClick(Sender: TObject);
  procedure CallGoalButtonClick(Sender: TObject);
  procedure ExitGoalButtonClick(Sender: TObject);

  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

The global variables "Main and "LocalString" declare variables for the main form, and the local buffer respectively.

```
Var
  Main : TMain;
  PrologID : Integer;
  LocalString : array [0..32768] of char; {local buffer for Prolog}

implementation

{$R *.DFM}
```

CLIENT1.PAS - The TMain.FormCreate Procedure

This procedure is called when the main form, "Form1" is created.

```
procedure TMain.FormCreate(Sender: TObject);
begin
```

An attempt is made to load ***WIN-PROLOG*** , and if that fails then a message is displayed and the application terminates:

```
PrologID := LoadProlog('',0,0,0) ;

if PrologID < 0 then
begin
 Application.MessageBox( PChar('LoadProlog Error '+ IntToStr( PrologID ) + '!'),
'Client Meals Example', 16 );
```

```
    Application.Terminate
end;
```

Provided ***WIN-PROLOG*** loaded correctly, a call is made via the *Intelligence Server* to load the example file, *CLIENT.PL*, which should be in the working directory:

```
    InitGoal( PrologID, 'consult(client). ' );
    CallGoal( PrologID );
    ExitGoal( PrologID );
end;
```

CLIENT1.PAS - The ClientCallGoal Procedure

This procedure wraps up two *Intelligence Server* functions, namely "CallGoal" and "TellGoal", providing user input from the "Choice" form whenever one of these functions returns output beginning with the letter "I" (for "input"):

```
{ Call the current goal, until Prolog needs no further input }

procedure ClientCallGoal( ID : Integer );
begin
  StrCopy( LocalString, CallGoal( ID ) );

  while( LocalString[0] = 'I' ) do
  begin
    GetUserInput( LocalString+8 );
    StrCopy( LocalString, TellGoal( ID, LocalString ) );
  end;


if ord(LocalString[0]) = 0 then
   Main.OutputMemo.Text :=  'NO PENDING QUERIES'
else
   Main.OutputMemo.Text := LocalString;
end;
```

The GetUserInput is called whenever Prolog requires more input.

CLIENT1.PAS The GetUserInput Procedure

```
{ Get the menu choice from the user }

procedure GetUserInput( Title : PChar );
begin
  Choice.Caption := StrPas( Title );
  if Choice.ShowModal = MB_OK then
    strPCopy( LocalString, Choice.Result )
  else
     strCopy( LocalString, 'any_meal. ' );
end;
```

CLIENT1.PAS - The TMain.FormDestroy Procedure

This is called when the main form is destroyed, and tells Prolog to halt.

```
procedure TMain.FormDestroy(Sender: TObject);
begin
  HaltProlog( PrologID );
  Application.Terminate;
end;
```

CLIENT1.PAS - The TMain.InitGoalButtonClick Procedure

This procedure is called when the "Init Goal" button is clicked. It initialises the goal "menu. " and then calls the procedure "ClientCallGoal".

```
procedure TMain.InitGoalButtonClick(Sender: TObject);
begin
  InitGoal( PrologID, 'menu. ' );
  ClientCallGoal( PrologID );
end;
```

CLIENT1.PAS - The TMain.CallGoalButtonClick Procedure

This procedure is called when the "Call Goal" button is clicked. It re-calls the current goal.

```
procedure TMain.CallGoalButtonClick (Sender: TObject);
begin
  Main.OutputMemo.Text := CallGoal( PrologID );
end;
```

CLIENT1.PAS - The TMain.ExitGoalButtonClick Procedure

This procedure is called when the user clicks on the "Exit Goal". The current goal is exited, and the previous goal is re-called.

```
procedure TMain.ExitGoalButtonClick(Sender: TObject);
begin
  ExitGoal( PrologID );
  ClientCallGoal( PrologID );
end;

end.
```

CLIENT2.PAS - The Input Form

The "client2" unit implements the input form "Choice".

```
unit client2;

{ Get the user users menu choice }

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

Type
  TChoice = class(TForm)
```

```
    AnyMeal: TButton;
    GoodMeal: TButton;
    CheapMeal: TButton;
    YuppieMeal: TButton;
    DietMeal: TButton;
    GluttonMeal: TButton;
    procedure AnyMealClick(Sender: TObject);
    procedure GoodMealClick(Sender: TObject);
    procedure CheapMealClick(Sender: TObject);
    procedure YuppieMealClick(Sender: TObject);
    procedure DietMealClick(Sender: TObject);
    procedure GluttonMealClick(Sender: TObject);

  private
    { Private declarations }

  public
    { Public declarations }
    Result : String;
  end;

 var
   Choice : TChoice;

implementation

{$R *.DFM}
```

CLIENT2.PAS - The TChoice.Click Procedures

When the user clicks on a button the Result string is set to the corresponding Prolog goal and the modal result is set to "mrOK", to end the modal dialog.

```
procedure TChoice.AnyMealClick(Sender: TObject);
begin
  Result := 'any_meal. ';
  ModalResult := mrOK;
end;

procedure TChoice.GoodMealClick(Sender: TObject);
begin
  Result := 'good_meal. ';
  ModalResult := mrOK;
end;

procedure TChoice.CheapMealClick(Sender: TObject);
begin
  Result := 'cheap_meal. ';
  ModalResult := mrOK;
end;

procedure TChoice.YuppieMealClick(Sender: TObject);
begin
```

```
  Result := 'yuppie_meal. ';
  ModalResult := mrOK;
end;

procedure TChoice.DietMealClick(Sender: TObject);
begin
  Result := 'diet_meal. ';
  ModalResult := mrOK;
end;

procedure TChoice.GluttonMealClick(Sender: TObject);
begin
  Result := 'glutton_meal. ';
  ModalResult := mrOK;
end;

end.
```

Chapter 6 The Java Interface

For the Java language the intelligence server is packaged as a class, which is an interface to the native code, *JINT386W.DLL*.

The Intelligence Server Class

The native code interface to the intelligence server is represented in the class *Int386w*. The functions will be accessed through an instance of this class.

```
public class Int386w
{
   private int Id;

public static int isERR_ENCODE = -1;
public static int isERR_TICKLE = -2;
public static int isERR_CMUTEX = -3;
public static int isERR_WMUTEX = -4;
public static int isERR_LOCATE = -5;
public static int isERR_CREATE = -6;
public static int isERR_MAPFIL = -7;
public static int isERR_MODULE = -8;
public static int isERR_PROLOG = -9;
public static int isERR_ACTIVE = -10;
public static int isERR_WINDOW = -11;



public int loadProlog( String s ) {
   return( loadProlog( s, 0, 0 ) );
}

public int loadProlog( String s, int desire, int tickle ) {
   Id = nativeLoadProlog( s, desire, tickle );
   return( Id );
}

public int haltProlog() {
   if( nativeHaltProlog(Id) != 0 ) {
        Id = 0;
        return 1;
  }  else
        return 0;
}

 public String initGoal( String Goal ) {
        return( nativeInitGoal( Id, Goal ) );
}
```

```
public String callGoal( ) {
        return( nativeCallGoal( Id ) );
}

 public String tellGoal( String Goal ) {
        return( nativeTellGoal( Id, Goal ) );
}

 public int exitGoal( ) {
   return( nativeExitGoal(Id) );
}

native int     nativeLoadProlog( String s, int desire, int tickle );
native int     nativeHaltProlog( int prolog );
native String        nativeInitGoal( int prolog, String goal );
native String        nativeCallGoal( int prolog );
native String        nativeTellGoal( int prolog, String goal );
native int     nativeExitGoal( int prolog );

static {
    System.loadLibrary( "jint386w" );
}

public void finalize()  {
  if( 0 != Id ) {
     haltProlog();
     Id = 0;
  }
}

}
```

JAVA CLIENT a worked example

The *Java CLIENT* example is divided into three files *Int386w.JAVA*, *Client.JAVA* and *InputDlg.JAVA*. The *Int386w.JAVA* file is the java file described previously. The following is a brief description of the other files.

CLIENT.JAVA

This is the main file for the example, and uses an instance of the Int386w class to talk to Prolog.

```
/*
** CLIENT, an example program to demonstrate the use of the Int386w
** class.
*/

import java.awt.*;
import java.awt.event.*;


 public class Client extends Frame
{
  Button      init_button, call_button, exit_button;
  TextArea    output_text;
  Int386w           prolog;

  public static void main(String[] main_args)
  {
   new Client();
  }

// Constructor
  public Client()
  {
    super( "CLIENT" );

    prolog = new Int386w();

    prolog.loadProlog( "" );
    prolog.initGoal( "consult('client'). " );
    prolog.callGoal();
    prolog.exitGoal();

    GridLayout grid = new GridLayout(3,1);
    Panel buttonPanel = new Panel( grid );
    Panel textPanel   = new Panel(new GridLayout(1,1));

    this.add( textPanel,   BorderLayout.CENTER );
    this.add( buttonPanel, BorderLayout.EAST );

    init_button = new Button( "Init Goal" );
    call_button = new Button( "Call Goal" );
```

```
  exit_button = new Button( "Exit Goal" );

  output_text = new TextArea();

  buttonPanel.add( init_button );
  buttonPanel.add( call_button );
  buttonPanel.add( exit_button );

  textPanel.add( output_text );

  this.addWindowListener( new WindowAdapter()
                {  public void windowClosing(WindowEvent e)
                   {  prolog.haltProlog();
                     System.exit(0);
                   }
                }
              );

  init_button.addActionListener( new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        prolog.initGoal( "menu. " );
        ClientCallGoal();
      }
    }
  );

  call_button.addActionListener( new ActionListener()
  {
      public void actionPerformed(ActionEvent e)
      {
        output_text.setText(prolog.callGoal());
      }
    }
  );

  exit_button.addActionListener( new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        prolog.exitGoal();
        ClientCallGoal();
      }
    }
  );

  this.pack();
  this.show();
}

private void ClientCallGoal()
{
```

```
      String Str = prolog.callGoal();

      InputDlg choice = new InputDlg( this, "Do you want meals of type:" );

      if( !Str.equals("") )
      while( Str.charAt(0) == 'T' )
      {
        Str = choice.execute();
        Str = prolog.tellGoal( Str );
      }

      if( Str.equals("") )
        output_text.setText("NO PENDING QUERIES" );
      else
        output_text.setText( Str );
    }
  }
```

INPUTDLG.JAVA

The Form2 class handles the user s menu choice.

```
  import java.awt.*;
  import java.awt.event.*;

  class InputDlg extends Dialog
  {
    Button any_button,
         good_button,
         cheap_button,
         yuppie_button,
         diet_button,
         glutton_button;

    Frame parent;

    public String response = "any_meal. ";
    Panel button_panel;

    InputDlg( Frame f, String title )
    {
     super( (Frame)f, title, true );
     any_button     = new Button( "Any Meal" );
     good_button    = new Button( "Good Meal" );
     cheap_button   = new Button( "Cheap Meal" );
     yuppie_button  = new Button( "Yuppie Meal" );
     diet_button    = new Button( "Diet Meal" );
     glutton_button = new Button( "Glutton Meal" );

     button_panel = new Panel();

     parent = f;
```

```
GridLayout  grid = new GridLayout(3,2);

setLayout(new BorderLayout(8,8) );

button_panel.setLayout( grid );

button_panel.add( any_button );
button_panel.add( good_button );
button_panel.add( cheap_button );
button_panel.add( yuppie_button );
button_panel.add( diet_button );
button_panel.add( glutton_button );

add("Center", button_panel );

any_button.addActionListener( new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        setResponse( "any_meal. " );
      }
    }
  );

good_button.addActionListener( new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        setResponse( "good_meal. " );
      }
    }
  );

cheap_button.addActionListener( new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        setResponse( "cheap_meal. " );
      }
    }
  );

yuppie_button.addActionListener( new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        setResponse( "yuppie_meal. " );
      }
    }
  );

diet_button.addActionListener( new ActionListener()
```

```
        {
          public void actionPerformed(ActionEvent e)
          {
            setResponse( "diet_meal. " );
          }
        }
      );

     glutton_button.addActionListener( new ActionListener()
        {
          public void actionPerformed(ActionEvent e)
          {
            setResponse( "glutton_meal. " );
          }
        }
      );

       this.pack();
  } // end InputDlg

  public void setResponse( String s )
  {
    response = s;
    this.setVisible(false);
  }

  public String execute ()
  {
    this.show();
    return response;
  }
}
```

Chapter 7 The Visual Basic 6 Interface

The *Intelligence Server* interface for Visual Basic 6 follows the same logic as the C server.

Using *VINT386W.DLL* in an Application

You can include the *INT386W.BAS* into a project by selecting the File/Add File... command in the Visual Basic 6 environment, this file declares the DLL function calls of the *VINT386W.DLL* and the error constants.

Loading Prolog

To load Prolog you call the function LoadProlog .

```
Id& = LoadProlog("",0,0,0)
```

The variable Id& will contain the result of the load, if the result is a positive integer it represents the identifier for an instance of Prolog. If the result is a negative integer then it represents an error.

If you want to load Prolog with command line switches, then pass this in the string parameter. The following statements load Prolog with a heap space of 512 and a text space of 1024:

```
Id& = LoadProlog("/H512 /T1024",0,0)
```

Halting Prolog

To halt Prolog call the sub-routine, HaltProlog with the instance ID.

```
HaltProlog( Id& )
```

The *WIN-PROLOG* session must be terminated using this procedure and not by running the Prolog predicate *halt/0*. This procedure takes no input and produces no output.

Initialising a Prolog Goal

To initialise a Prolog goal prior to calling it, you call the function InitGoal:.

```
Result$ = InitGoal( Id&, "write( Hello World ). " )
```

Please note that all goals submitted should be terminated with a full-stop and a whitespace character, so as to be read correctly by Prolog. The only thing of interest returned from InitGoal is error messages, such as syntax errors.

Calling a Prolog Goal

To call the currently active Prolog goal you call the function CallGoal with the instance ID .

```
Result$ = CallGoal( Id& )
```

Calling the function CallGoal will return the output from running (or backtracking into) the current Prolog goal for the given instance. The string returned has the form:

```
XSYYYYNL
```

Where:

```
X = T->true, F->fail, E->error, I->input
S = space
YYYY = four-digit query number (0000 is the first query)
NL = new line (Chr$(13)+Chr$(10))
```

Immediately following this header is either any output generated by the called goal or an input prompt.

Examples of calls are:

```
Result$ = CallGoal( Id& )
Result$ = mid$( CallGoal( Id& ), 9)
```

The first example simply sets a result string to be the complete string returned from the server. The second example skips past the header and sets the return string to just the output from the goal.

You can call the CallGoal function repeatedly to force the current goal to backtrack and return successive solutions.

Returning Information to a Currently Running Prolog Goal

To return input to a Prolog goal, after it has requested input, you call the function TellGoal:

```
Result$ = TellGoal( Id&, "Input String" )
```

This function is used to provide input to a running query, the returned string may itself be a request for further input: this function is the only valid response to an input request. The following statement returns the information "This is your input" to the query currently requesting input:

```
TellGoal( Id&, "This is your input" )
```

Exiting a Prolog Goal

To exit the current Prolog goal, you call the sub-routine ExitGoal. This sub-routine is used to terminate a query, once no further solutions are required. It takes the instance identifier and returns no output. The following statement exits the currently active query:

```
ExitGoal( Id& )
```

After this call, the next most recently used query becomes active once more.

CLIENT.EXE - A Worked Example

The remainder of this chapter shows how the properties described above are put into practice in a simple Visual Basic 6 example that uses the *INT386W.DLL* to drive the *CLIENT.PL* program. This program is simply a teletype-only version of the *MEALS.PL* example shipped with **WIN-PROLOG** . The *CLIENT.EXE* program, which can be found in the *SERVER\VB* sub-directory of the **WIN-PROLOG** home directory, is built from four files, which are described in the following sections.

CLIENT.VBP - the Project File

This file loads the *CLIENT.EXE* application files into the Visual Basic 6 environment.

INT386W.BAS - Global Definitions

This file contains the *VINT386W.DLL* function declarations:

```
Attribute VB_Name = "prolog"

#Const LPA_UNICODE = 0
Public Const ERR_ENCODE = -1
Public Const ERR_TICKLE = -2
Public Const ERR_CMUTEX = -3
Public Const ERR_WMUTEX = -4
Public Const ERR_LOCATE = -5
Public Const ERR_CREATE = -6
Public Const ERR_MAPFIL = -7
Public Const ERR_MODULE = -8
Public Const ERR_PROLOG = -9
Public Const ERR_ACTIVE = -10
Public Const ERR_WINDOW = -11

'Declaration of VINT386W.DLL SKIN interface

' If the constant LPA_UNICODE is declared as 1
#If LPA_UNICODE = 1 Then
Private Declare Function LoadP Lib "VINT386W" Alias "vLoadProlog" (ByVal Command
As String, ByVal Desire As Integer, ByVal Encode As Integer, ByVal Tickle As Integer)
As Long
Public Declare Sub HaltProlog Lib "VINT386W" Alias "vHaltProlog" (ByVal Prolog As
Long)
Public Declare Function InitG Lib "VINT386W" Alias "vInitGoalW" (ByVal Prolog As
Long, Buffer As Any, ByVal size As Long) As Long
Public Declare Function CallG Lib "VINT386W" Alias "vCallGoalW" (ByVal Prolog As
Long, Buffer As Any, ByVal size As Long) As Long
Public Declare Function TellG Lib "VINT386W" Alias "vTellGoalW" (ByVal Prolog As
Long, Buffer As Any, ByVal size As Long) As Long
Public Declare Sub ExitGoal Lib "VINT386W" Alias "vExitGoal" (ByVal Prolog As Long)
Dim Buffer() As Byte
Dim size As Long

Public Function InitGoal(ByVal Prolog As Long, ByVal Goal As String) As String
  ReDim Buffer(1 + LenB(Goal))
  Buffer() = Goal
  Count = InitG(Prolog, Buffer(0), Len(Goal))
```

```
  InitGoal = Left(Buffer, Count)
End Function
Public Function CallGoal(ByVal Prolog As Long) As String
  ReDim Buffer(size)
  Count = CallG(Prolog, Buffer(0), size)
  CallGoal = Left(Buffer, Count)
End Function
Public Function TellGoal(ByVal Prolog As Long, Goal As String) As String
  ReDim Buffer(1 + LenB(Goal))
  Buffer() = Goal
  Count = TellG(Prolog, Buffer(0), Len(Goal))
  TellGoal = Left(Buffer, Count)
End Function

Public Function LoadProlog(Command As String, Optional ByVal Desire As Integer =
0, Optional ByVal Encoding = 1, Optional ByVal Tickle = 0) As Integer
  If Desire = 0 Then
    size = 65535
  Else
    size = Desire
  End If
  LoadProlog = LoadP(Command, Desire, Encoding, Tickle)
End Function

'If the constant LPA_UNICODE is not declared as 1
#Else

Private Declare Function LoadP Lib "VINT386W" Alias "vLoadProlog" (ByVal Command
As String, ByVal Desire As Integer, ByVal Encode As Integer, ByVal Tickle As Integer)
As Long
Public Declare Sub HaltProlog Lib "VINT386W" Alias "vHaltProlog" (ByVal Prolog As
Long)
Public Declare Function InitGoal Lib "VINT386W" Alias "vInitGoalA" (ByVal Prolog As
Long, ByVal StartGoal As String) As String
Public Declare Function CallGoal Lib "VINT386W" Alias "vCallGoalA" (ByVal Prolog As
Long) As String
Public Declare Function TellGoal Lib "VINT386W" Alias "vTellGoalA" (ByVal Prolog As
Long, ByVal TellGoal As String) As String
Public Declare Sub ExitGoal Lib "VINT386W" Alias "vExitGoal" (ByVal Prolog As Long)

Public Function LoadProlog(ByVal Command As String, Optional ByVal Desire As
Integer = 0, Optional ByVal Encoding = 0, Optional ByVal Tickle = 0) As Integer
  LoadProlog = LoadP(Command, Desire, Encoding, Tickle)
End Function

#End If
```

CLIENT1.FRM - The Client Example Main Form

This file defines the main dialog for the example application and its handler. The function that loads the *WIN-PROLOG* system into memory, followed by the *CLIENT.PL* example program is also defined here. This main dialog handles messages from three buttons, corresponding to the actions "Init Goal", "Call

Goal" and "Exit Goal", and displays output from calls to *CLIENT*, which is running on the *Intelligence Server*. A secondary dialog, defined in *CLIENT2.FRM* is used when necessary to elicit user input.

CLIENT1.FRM - The Form_Load Sub-routine

This sub-routine is the initial entry point for the example application, and is responsible for loading the ***WIN-PROLOG*** system into memory. If the load fails a message box appears informing the user and the application then ends.

```
This declares the variable that will become the instance identifier for this example
Dim Prolog As Long

Private Sub Form_Load()

  Prolog = LoadProlog("")
  If Prolog < 0 Then
    MsgBox "LoadProlog Error " & Str(Prolog) & "!", vbCritical
    End
  End If
```

Provided ***WIN-PROLOG*** loaded correctly, a call is made via the *Intelligence Server* to load the example file, *CLIENT.PL*, which should be in the current working directory:

```
'Consult the meals example
'by initialising the goal,
'calling the goal
'and exiting the goal
  X$ = InitGoal(Prolog, "consult(client). ")
  X$ = CallGoal(Prolog)
  ExitGoal Prolog

End Sub
```

CLIENT1.FRM - The client_call_goal Sub-routine

This sub-routine calls and then processes any input requests from the currently running goal. The while loop tests for the "I" character at the beginning of the output from the called goal. When this character is present the goal is requesting input so the input text contained in the output from the called goal is set as the caption for the "PrologInput" dialog. This dialog is then run and the result is sent back to the waiting goal using the "TellGoal" action. If the goal then requests more input, the while loop will repeat, otherwise the loop and the sub-routine will finish.

```
Public Sub client_call_goal()

'Call (or backtrack into) the current goal
  OutString = CallGoal(Prolog)

'While Prolog requires more input from the user
  While Mid$(OutString, 1, 1) = "I"
  InString = get_user_input(Mid$(OutString, 9))
  OutString = TellGoal(Prolog, InString)
Wend
```

```
'If the returned string from Prolog is empty
'then there are no further queries
  If OutString = "" Then
    Output.Text = "NO PENDING QUERIES"
  Else
    Output.Text = OutString
  End If

End Sub
```

CLIENT1.FRM - The Button Click Handlers

The following sub-routine handles click messages from the "InitGoalButton" button, which is labelled "Init Goal":

```
Private Sub InitGoalButton_Click()

   'Initialise the "menu. " goal
   X$ = InitGoal(Prolog, "menu. ")

   'And call the goal
  client_call_goal

End Sub
```

This sub-routine initialises the menu. goal and then calls the client_call_goal sub-routine to call the goal.

The following sub-routine handles click messages from the "CallGoalButton" button, labelled "Call Goal":

```
Private Sub CallGoalButton_Click ()

  Call ( or backtrack into ) a goal
   Output.Text = CallGoal

End Sub
```

This sub-routine calls the "CallGoal" function to call the current Prolog query.

```
Private Sub CallGoalButton_Click()

 'Call ( or backtrack into) a goal
  Output.Text = CallGoal(Prolog)

End Sub
```

The following sub-routine handles click messages from the "ExitGoalButton" button, labelled "Exit Goal":

```
Private Sub ExitGoalButton_Click()
'Exit the current goal
  ExitGoal Prolog
'and redo the previously stacked goal
```

```
    client_call_goal
End Sub
```

This sub-routine exits the currently active goal and then calls the client_call_goal sub-routine to call the previously active goal (if any).

CLIENT1.FRM - The get_user_input Sub-routine

The following function handles input from the user:

```
Public Function get_user_input(Title$) As String
'Get user choice from the input dialog
   PrologInput.Caption = Title$
   PrologInput.Show 1
   get_user_input = PrologInput.Result
End Function
```

CLIENT1.FRM - The Form_Unload Sub-routine

When the form unloads, as happens after the user clicks on the close-box, this sub-routine closes down the *Intelligence Server* by calling the "HaltProlog" procedure and then terminates the application.

```
Private Sub Form_Unload(Cancel As Integer)
' Unload Prolog
  HaltProlog Prolog
  End
End Sub
```

CLIENT2.FRM - The Client Example Input Form

This form creates a simple dialog, containing six command buttons that prompts the user for input and sets a property to the result.

```
Dim Output$

Private Sub any_meal_Click()
        Result = "any_meal. "
        PrologInput.Hide
End Sub

Private Sub CheapMeal_Click()
        Result = "cheap_meal. "
        PrologInput.Hide
End Sub

Private Sub Diet_Meal_Click()
        Result = "diet_meal. "
        PrologInput.Hide
End Sub

Private Sub Glutton_Meal_Click()
        Result = "glutton_meal. "
```

```
        PrologInput.Hide
End Sub

Private Sub GoodMeal_Click()
        Result = "good_meal. "
        PrologInput.Hide
End Sub

Private Sub Yuppie_Meal_Click()
        Result = "yuppie_meal. "
        PrologInput.Hide
End Sub

Public Property Get Result() As String
        Result = Output$
End Property

Public Property Let Result(vNewValue As String)
        Output$ = vNewValue
End Property
```

Chapter 8 The .NET interface

Microsoft s new SDK platform .NET imposes special constraints upon the intelligence server, as the .NET code runs within a managed environment on a virtual machine. Therefore an interface has to be defined between the managed code and the intelligence server unmanaged code. This has been created as LPA.DLL, which provides a class wrapper to the native intelligence server function calls. The source code for this DLL is included in the INCLUDES directory in your installed prolog directory, you may need to use this code to recompile the DLL under whichever .NET framework you are working with.

The wrapper class is defined as LPA.IntServer and the methods the class exposes in C# are:

The constructor overrides the normal constructor so that creating a new IntServer object you can specify if it will handle Unicode or Ansi string data as:

```
LPA.IntServer myProlog  = new LPA.IntServer("",0,1,0) ;
```

Where the arguments are:

The commandline can take switches etc.

The buffersize specifies the size of the interface buffer, 0 represents a buffer size of 64kb.

The codestyle can be 0 or 1, 1 specifies a Unicode interface and 0 an Ansi interface, thereafter all string references to that object will be in that type.

The tickle can be 0 or 1, 0 means don t tickle Prolog, 1 means tickle Prolog to keep it active.

The class does not define LoadProlog nor HaltProlog, LoadProlog is run when a new instance is created and HaltProlog will be executed when the instance is garbage collected.

If an error occurs in the creation of an object then a PrologException will be thrown, and the exception message will contain details of the error.

```
public string InitGoal( string Goal );
public string CallGoal();
public string TellGoal( string TellGoal );
public int ExitGoal();
```

The equivalent definitions in VB.NET would be:

```
Public Function InitGoal( ByVal Goal As String ) As String
Public Function CallGoal() As String
Public Function ExitGoal() As Int32
Public Function TellGoal( ByVal Goal As String ) As String )
```

You can add the LPA.IntServer class to your project through the *Project/Add Reference…* menu of the *Visual Studio.NET* environment. Use the *browse* button of the *.NET* page of the *Add Reference* dialog to select the LPA.DLL it will then be copied when required into the local build directory (*debug* or *release*). You will also need to make the INT386W.DLL visible to the LPA.DLL either by putting it (and the INT386W.SYS, INT386W.OVL and Prolog source code) into an appropriate directory or by making them visible on the path. An appropriate directory can be the *debug* directory of your project. Because .NET

managed applications compile down into a single virtual machine, only one version of the INT386W.DLL is required for all languages. After you have done that, using the Intelligence Server is very straightforward.

Setting up projects in Visual Studio 2008 and 2010

- Setup a new project.
- Add the required include files to the project for your language.
- Add the generic Intelligence Server files to your project, namely, **INT386W.DLL**, **INT386W.OVL** and **INT386W.SYS**.
- For each of those files right-click and get properties, in the properties dialog select the option **Copy to Output Directory : Copy always**.
- Do the same for any other prolog or flex programs, including the **flex.pc** if needed.

This will ensure that the required files are copied to the build directory so that you can run the application from within Visual Studio.

Examples

This section explains an example for C# the same principles are applicable to VB.NET. To set up the disk examples copy them to your projects directory and make sure that INT386W.DLL, INT386W.OVL, INT386W.SYS and CLIENT.PL are in an appropriate directory. The INT386W.DLL can be found in the server\examples\c folder.

C# worked example

Form1.CS, this is the main dialog code.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace csclient
{
   public class Form1 : System.Windows.Forms.Form
   {
```

Declare a new Intelligence server object

```
      private LPA.IntServer prolog;
      private System.Windows.Forms.TextBox textBox1;
      private System.Windows.Forms.Button bInitGoal;
      private System.Windows.Forms.Button bCallGoal;
      private System.Windows.Forms.Button bExitGoal;

      private System.ComponentModel.Container components = null;

      public Form1()
```

```
{
        //
        // Required for Windows Form Designer support
        //
        InitializeComponent();
}

protected override void Dispose( bool disposing )
{
        if( disposing )
        {
                if (components != null)
                {
                        components.Dispose();
                }
        }
        base.Dispose( disposing );
}

private void InitializeComponent()
{
```

Setting up the controls on the form.

```
        this.textBox1 = new System.Windows.Forms.TextBox();
        this.bInitGoal = new System.Windows.Forms.Button();
        this.bCallGoal = new System.Windows.Forms.Button();
        this.bExitGoal = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // textBox1
        //
        this.textBox1.Location = new System.Drawing.Point(8, 8);
        this.textBox1.Multiline = true;
        this.textBox1.Name = "textBox1";
        this.textBox1.ScrollBars = System.Windows.Forms.ScrollBars.Both;
        this.textBox1.Size = new System.Drawing.Size(280, 264);
        this.textBox1.TabIndex = 0;
        this.textBox1.Text = "";
        //
        // bInitGoal
        //
        this.bInitGoal.Location = new System.Drawing.Point(296, 8);
        this.bInitGoal.Name = "bInitGoal";
        this.bInitGoal.TabIndex = 1;
        this.bInitGoal.Text = "Init Goal";
        this.bInitGoal.Click += new System.EventHandler(this.bInitGoal_Click);
        //
        // bCallGoal
        //
        this.bCallGoal.Location = new System.Drawing.Point(296, 40);
        this.bCallGoal.Name = "bCallGoal";
```

```
            this.bCallGoal.TabIndex = 1;
            this.bCallGoal.Text = "Call Goal";
            this.bCallGoal.Click += new System.EventHandler(this.bCallGoal_Click);
            //
            // bExitGoal
            //
            this.bExitGoal.Location = new System.Drawing.Point(296, 72);
            this.bExitGoal.Name = "bExitGoal";
            this.bExitGoal.TabIndex = 1;
            this.bExitGoal.Text = "Exit Goal";
            this.bExitGoal.Click += new System.EventHandler(this.bExitGoal_Click);
            //
            // Form1
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(384, 286);
            this.Controls.AddRange(new System.Windows.Forms.Control[] {
                        this.bInitGoal,
            this.textBox1,
   this.bCallGoal,                                                    this.bExitGoal});
            this.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedDialog;
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "Form1";
            this.StartPosition =
System.Windows.Forms.FormStartPosition.CenterScreen;
            this.Text = "Client";
            this.Load += new System.EventHandler(this.Form1_Load);
            this.ResumeLayout(false);

      }

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
            Application.Run(new Form1());
      }
```

When the form is loaded, a new LPA.IntServer object is created in this case with the unicode style, this automatically loads Prolog, if an error occurs then an exception will be thrown. Otherwise the prolog source code for the application is loaded into the Prolog object.

```
private void Form1_Load(object sender, System.EventArgs e)
      {
            try
            {
                  prolog = new LPA.IntServer("",0,1,0);
                  prolog.InitGoal( "load_files(prolog(client)). " );
```

```
                prolog.CallGoal();
                prolog.ExitGoal();
        }
        catch( Exception except )
        {
                System.Windows.Forms.MessageBox.Show( except.Message );
                Application.Exit();
        }
}
```

CallGoal will call an initialized goal repeatedly until Prolog requires no more input.

```
private void CallGoal()
{
        String s = prolog.CallGoal();
        Form2  question = new Form2();

        if( s == "" )
        {
                textBox1.Text = "No pending queries";
        }
        else
        {
                while( s.StartsWith("I") )
                {
                        question.Text = s.Substring(8);
                        question.ShowDialog(this);

                        s = prolog.TellGoal( question.answer );
                }
                textBox1.Text = s;
        }
}
```

On bInitGoal clicked the goal "menu. " is initialized and it's first call is made.

```
private void bInitGoal_Click(object sender, System.EventArgs e)
{
        textBox1.Text = prolog.InitGoal( "menu. " );

        CallGoal();
}
```

On bCallGoal clicked the initialized goal is called.

```
private void bCallGoal_Click(object sender, System.EventArgs e)
{
        CallGoal();
}
```

On bExitGoal clicked the current goal is exited and any previous stacked goal is called.

```
        private void bExitGoal_Click(object sender, System.EventArgs e)
        {
            prolog.ExitGoal();
            CallGoal();
        }
    }
}
```

Form2.CS this is used to choose the type of meal.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace csclient
{
```

Form2 is a modal dialog which allows the user to select different meal types for Prolog to generate menus for.

```
    public class Form2 : System.Windows.Forms.Form
    {
        public string answer;

        private System.Windows.Forms.Button bAnyMeal;
        private System.Windows.Forms.Button bGoodMeal;
        private System.Windows.Forms.Button bCheapMeal;
        private System.Windows.Forms.Button bYuppieMeal;
        private System.Windows.Forms.Button bDietMeal;
        private System.Windows.Forms.Button bGluttonMeal;
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public Form2()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if(components != null)
```

```
                {
                        components.Dispose();
                }
        }
        base.Dispose( disposing );
}

private void InitializeComponent()
{
        this.bAnyMeal = new System.Windows.Forms.Button();
        this.bGoodMeal = new System.Windows.Forms.Button();
        this.bCheapMeal = new System.Windows.Forms.Button();
        this.bYuppieMeal = new System.Windows.Forms.Button();
        this.bDietMeal = new System.Windows.Forms.Button();
        this.bGluttonMeal = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // bAnyMeal
        //
        this.bAnyMeal.Location = new System.Drawing.Point(8, 8);
        this.bAnyMeal.Name = "bAnyMeal";
        this.bAnyMeal.Size = new System.Drawing.Size(88, 23);
        this.bAnyMeal.TabIndex = 0;
        this.bAnyMeal.Text = "Any Meal";
        this.bAnyMeal.Click += new System.EventHandler(this.bAnyMeal_Click);
        //
        // bGoodMeal
        //
        this.bGoodMeal.Location = new System.Drawing.Point(8, 40);
        this.bGoodMeal.Name = "bGoodMeal";
        this.bGoodMeal.Size = new System.Drawing.Size(88, 23);
        this.bGoodMeal.TabIndex = 0;
        this.bGoodMeal.Text = "Good Meal";
        this.bGoodMeal.Click += new
System.EventHandler(this.bGoodMeal_Click);
        //
        // bCheapMeal
        //
        this.bCheapMeal.Location = new System.Drawing.Point(8, 72);
        this.bCheapMeal.Name = "bCheapMeal";
        this.bCheapMeal.Size = new System.Drawing.Size(88, 23);
        this.bCheapMeal.TabIndex = 0;
        this.bCheapMeal.Text = "Cheap Meal";
        this.bCheapMeal.Click += new
System.EventHandler(this.bCheapMeal_Click);
        //
        // bYuppieMeal
        //
        this.bYuppieMeal.Location = new System.Drawing.Point(120, 8);
        this.bYuppieMeal.Name = "bYuppieMeal";
        this.bYuppieMeal.Size = new System.Drawing.Size(88, 23);
        this.bYuppieMeal.TabIndex = 0;
        this.bYuppieMeal.Text = "Yuppie Meal";
```

```
            this.bYuppieMeal.Click += new
System.EventHandler(this.bYuppieMeal_Click);
            //
            // bDietMeal
            //
            this.bDietMeal.Location = new System.Drawing.Point(120, 40);
            this.bDietMeal.Name = "bDietMeal";
            this.bDietMeal.Size = new System.Drawing.Size(88, 23);
            this.bDietMeal.TabIndex = 0;
            this.bDietMeal.Text = "Diet Meal";
            this.bDietMeal.Click += new System.EventHandler(this.bDietMeal_Click);
            //
            // bGluttonMeal
            //
            this.bGluttonMeal.Location = new System.Drawing.Point(120, 72);
            this.bGluttonMeal.Name = "bGluttonMeal";
            this.bGluttonMeal.Size = new System.Drawing.Size(88, 23);
            this.bGluttonMeal.TabIndex = 0;
            this.bGluttonMeal.Text = "Glutton Meal";
            this.bGluttonMeal.Click += new
System.EventHandler(this.bGluttonMeal_Click);
            //
            // Form2
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(218, 110);
            this.Controls.AddRange(new System.Windows.Forms.Control[] {
                        this.bAnyMeal,
                        this.bGoodMeal,
                        this.bCheapMeal,
                              this.bYuppieMeal,
                                    this.bDietMeal,
                                    this.bGluttonMeal});
            this.FormBorderStyle =
System.Windows.Forms.FormBorderStyle.FixedDialog;
            this.MaximizeBox = false;
            this.MinimizeBox = false;
            this.Name = "Form2";
            this.StartPosition =
System.Windows.Forms.FormStartPosition.CenterParent;
            this.Text = "Form2";
            this.Load += new System.EventHandler(this.Form2_Load);
            this.ResumeLayout(false);

        }
```

For each button click the appropriate answer is created and the dialog terminates with success.

```
        private void bAnyMeal_Click(object sender, System.EventArgs e)
        {
           answer="any_meal. ";
            this.DialogResult = DialogResult.OK;
        }
```

```
        private void bGoodMeal_Click(object sender, System.EventArgs e)
        {
            answer="good_meal. ";
            this.DialogResult = DialogResult.OK;

        }

        private void bCheapMeal_Click(object sender, System.EventArgs e)
        {
            answer="cheap_meal. ";
            this.DialogResult = DialogResult.OK;
        }

        private void bYuppieMeal_Click(object sender, System.EventArgs e)
        {
            answer="yuppie_meal. ";
            this.DialogResult = DialogResult.OK;

        }

        private void bDietMeal_Click(object sender, System.EventArgs e)
        {
            answer="diet_meal. ";
            this.DialogResult = DialogResult.OK;

        }

        private void bGluttonMeal_Click(object sender, System.EventArgs e)
        {
            answer="glutton_meal. ";
            this.DialogResult = DialogResult.OK;

        }
```

On loading the form the default answer of "any_meal. " is set.

```
        private void Form2_Load(object sender, System.EventArgs e)
        {
            answer="any_meal. ";
        }

    }
}
```

Chapter 9 The Python interface

The Python interface to the intelligence Server is the module INT386W.PY, declaring the class Int386w.

To use the module simply import it, as:

```
import Int386w
```

The Class Int386w comprises the following member functions:

- __init__ which is called when a Int386w object is created, this loads the prolog DLL
- __iter__ Returns the object as an iterator
- loadProlog
  - Loads Prolog with the given arguments
- initGoal
  - Initialise a goal for calling
- callGoal
  - Call or recall a previously initialised goal
- tellGoal
  - Return information in response to a request for input from prolog
- exitGoal
  - Exit a previously initialised goal.
- haltProlog

  Halt the current instance of Prolog, the same object can reload prolog using the loadProlog member function.
- next
  - This returns results from a previously initialised goal as requested, ending if a failure occurred.

In addition the module defines the PrologError Exception class that is raised if the prolog engine cannot be loaded for any reason.

The CLIENT.PY example

The client example is a command line program (as opposed to a gui example) which is simply run as a normal python program. To assemble the example, in a new directory copy from the<prolog>/SERVER/INCLUDES directory the files:

- INT386W.OVL
- INT386W.DLL
- INT386W.SYS
- INT386W.PY
- CLIENT.PL

and from the <prolog>/SERVER/EXAMPLES/PYTHON/CLIENT directory the file :

- CLIENT.PY

You can then run the example, assuming that the python interpreter is on the path by entering at a command line in the directory you created:

```
client.py
```

or maybe:

```
python client.py
```

Using the int386w module, the client example is :

```
import int386w

prolog = int386w.int386w()

prolog.loadProlog()

prolog.initGoal( "consult(client). " )

prolog.callGoal()

prolog.exitGoal()

# client.pl is loaded so start:

prolog.initGoal( "menu. " )

# our choices of yummy meals.

choices = { 'A':"any_meal. ",

            'B':"good_meal. ",

            'C':"cheap_meal. ",

            'D':"yuppie_meal. ",
```

```
        'E':"diet_meal. ",
        'F':"glutton_meal. "
        }
end_loop = 'Y'
while end_loop == 'Y':
  result = prolog.callGoal()

  r,c,s  = result[0], result[2:6], result[8:]
  if r == 'T':  # prolog has requested input
    print "Bonjour! Choose from one of our delicious menus"
    for key in choices :
      print key,choices[key]
    answer = raw_input("Press selection key, then <ENTER> to select your menu, or
any other to quit ").upper()
    try :
      choice = choices[answer]
    except :
      end_loop="Stop"
    else:
      prolog.tellGoal(choice)
  elif r == 'F': # The goal has finished (failed)
    print "\nNo more meals!\n"
    prolog.initGoal( "menu. " )
  elif r == 'T': # The goal has succeeded
    print s
    if raw_input("More .. (Q to quit)<ENTER>").upper() == 'Q':
      end_loop = "Stop"
  elif r == 'E': # An error occurred
    print result
```

```
    end_loop = "Stop"
print "Au revoir! Come again soon!"
prolog.exitGoal()
prolog.haltProlog()
```

The INT386W.PY module

```
import ctypes
class PrologError( Exception ) :
  """ Exception raised for errors in prolog.
      value : The numeric value of the error
      message : An arbitrary message explaining the error
  """
  def __init__( self, value, message ):
    self.value = value
    self.message = message

  def __str__( self ) :
    return repr(self.value)

class int386w() :
"""Intelligence Server Interface
    loadProlog( self, command, buffer, type )
    initGoal( self, goal )
    callGoal( self )
    exitGoal( self )
    tellGoal( self, data )
    haltProlog( self )
  """
```

```
def __init__(self) :
    "Load the Intelligence Server DLL."
    self.prolog = ctypes.windll.LoadLibrary( "Int386W.DLL" )

def __iter__( self ) :
  "Return an iterator."
  return self

def loadProlog( self, command="", buffer=0, type=0 ) :
  "Load prolog with the given arguments."
  result = self.prolog.LoadProlog( command, buffer, type, 0 )
  if result < 1 :
    raise PrologError( result, "An error occured while attempting to load prolog." )
  else:
    self.id = result
    return self.id

def initGoal( self, goal ) :
  "Initialize a goal ready for execution."
  return ctypes.c_char_p( self.prolog.InitGoal( self.id, goal ) ).value

def callGoal( self ) :
  "Execute a previously initialized goal."
  return ctypes.c_char_p( self.prolog.CallGoal(self.id) ).value

def exitGoal( self ) :
  "Drop or exit a previously initialized goal."
  return self.prolog.ExitGoal(self.id)
```

```
    def tellGoal( self, data ) :
      "Respond to a request for input from an executing goal."
      return ctypes.c_char_p( self.prolog.TellGoal( self.id, data ) ).value

    def haltProlog( self ) :
      "Halt the currently running prolog."
      return self.prolog.HaltProlog(self.id)

    def next( self ) :
      "Get the next result through backtracking, finish if failure reported"
      result = self.callGoal()
      r = result[0]
      if r == 'F':
        self.exitGoal()
        raise StopIteration
      else :
        return result

if __name__ == "__main__":

    prolog=int386w()
    prolog.loadProlog()
    prolog.initGoal( "statistics. ")

    s = prolog.callGoal()
    prolog.exitGoal()

    (r,c,v) = (s[0],s[2:6],s[8:])
    print r,c,v
```

```
prolog.initGoal( "member(P,[hi,this,is,prolog]), write(P). ")


r = 'T'
while( r == 'T' ) :
  s = prolog.callGoal()
  (r,c,v) = (s[0],s[2:6],s[8:])
  print r,c,v


prolog.exitGoal()


prolog.initGoal( "member(P,[hi,this,is,a,prolog,iterator]), write(P)" )


for s in prolog :
  (r,c,v) = (S[0], s[2:6], s[8:] )
  print r,c,v


prolog.haltProlog()
```

Chapter 10 Programming using the Intelligence Server

This chapter gives a brief overview of some of the techniques and pragmatics that can be used when developing an Intelligence Server application.

Returning variable bindings from Prolog calls

Because the Intelligence Server interface only returns the textual output from running a Prolog goal, a slight addition needs to be made to the calling of a program

Suppose you called the goal:

```
append( X, Y, [a,b,c,d,e] ).
```

This goal on its own has no textual output, it simply binds X and Y to the respective solutions. If you want to get back the results of the call, without changing the program being called, you need to output the terms as well. The following goal does this, as well as outputting each term on a new line:

```
append( X, Y, [a,b,c,d,e] ), write(X), nl, write(Y), nl.
```

If you wanted to keep the variable names associated with their results you could call the following goal:

```
append( X, Y, [a,b,c,d,e] ), write( 'X'=X ), nl, write( 'Y'=Y ), nl.
```

Returning individual elements of a solution

Sometimes the results of Prolog programs come back as structured terms. Instead of writing code in your front-end program to decompose these terms, it is often done more easily and efficiently by Prolog itself. For example if a Prolog program output a Prolog list, you could deconstruct that list into individual elements using *member/2*. The following goal outputs a sorted version of the input list:

```
sort([b,c,a,g,d,f,e],X), write(X), nl.
```

where the output is

```
[a,b,c,d,e,f,g]
```

You could re-arrange this call to return each element individually on backtracking using *member/2*:

```
sort([b,c,a,g,d,f,e],X), member(Y,X), write(Y), nl.
```

Debugging the *WIN-*PROLOG application code

The first stage is to write and debug the ***WIN-*PROLOG** code for the application. The ***WIN-*PROLOG** code for an application generally consists of pure textual input and output. From this point of view it is simple to test this code at the development stage from the ***WIN-*PROLOG** command-line.

Defining an *input/2* program

When a Prolog goal requires more information from the front-end, via the Intelligence Server, it does so using the goal *input/2*, as described in Chapter 1 under the heading *The TellGoal(int ID, LPSTR Input) Function*.

When debugging your Prolog code in isolation from the Intelligence Server and the front-end program, you need to define *input/2*. The following definition for *input/2*, is a simple and useful aid that takes a request for information and outputs it to the console and then initiates a read of the console to get the required information to pass back:

```
input( X, Y ) :-
  write( X ), nl,
  fread( s, 0, 0, Y ).
```

We can see this program in action by loading the Intelligence Server example *CLIENT.PL*.

```
?- ensure_loaded( prolog('SERVER\C\CLIENT.PL') ).
```

To test this program, compile the above definition of the *input/2* program and then enter the following goal at the ***WIN-PROLOG*** command-line:

```
?- repeat, menu, nl, write('<esc> to finish'), nl, getb( X ), X = 27.
```

Running this goal will prompt you for input, as follows:

```
?- repeat, menu, nl, write('<esc> to finish'), nl, getb( X ), X = 27.
Do you want meals of type:
|:
```

You can then respond by typing one of the inputs expected by the *CLIENT.PL* program:

```
|: diet_meal.
Today's delicious choice of diet_meal menus
Starter:              parma_ham_with_melon
   -oOo-
Main Course:      dover_sole
   -oOo-
Dessert:      fresh_fruit_salad
   -oOo-
Wine:         muscadet_sur_lie

price:  £24.43      calories:     625

<esc> to finish
```

At this point hitting any key, other than the <esc> key, will give another solution.

Testing the Prolog code in *WIN-*PROLOG via the Intelligence Server

You can test your Prolog application from the other end of the Intelligence Server using the following predicate definitions. To run these predicates you need to have made a temporary directory, say C:\TEMP1S and copied the following files:*PRO386W.OVL and PRO386W.EXE from <your prolog installation> directory and, INT386W.SYS, INT386W.DLL, INT386W.OVL, CLIENT.PL and PROLOG.PL* from the <your prolog installation>*\SERVER\INCLUDES* directory into this directory. Please note that *PROLOG.PL* is a source code version of the following definitions for your convenience, so that you don t have to type them yourself.

First of all run the *PRO386W.EXE* program from the temporary directory, then open and compile the file *PROLOG.PL*.

The first predicate you need to run is load_dll/0, this will load the dll from the current prolog directory. You only need to run this once at the beginning of the session. You could remember the handle from the DLL so that you can shut it down later, however, it is not needed in this case, as the DLL will be shut down when you quit your prolog session.

```
load_dll :-
  absolute_file_name( prolog(  int386w.dll  ), Dll ),
  atom_string( Dll, Dllstr ),
  winapi( (kernel32, LoadLibraryA ), [Dllstr], 0, _ ) .
```

The following predicate loads another instance of prolog from the working prolog through the *Intelligence server* interface, you should only call this once, or after a halt_prolog/1 :

```
load_prolog( Id ) :-
  load_prolog( ‘ ‘, 0, 0, 0, Id ).

load_prolog( Commandswitches, Buffersize, Encoding, Tickle, Id ) :-
  winapi( (int386w,'LoadProlog'), [Commandswitches,Buffersize,Encoding,Tickle], 0,
Result ),
  Id = Result.
```

A negative value returned in Id indicates a problem has occurred in attempting to load the server prolog, you should then check that you have all the correct files in the correct place. As Prolog is running as a DLL you should also check on the task-manager that you do not have a zombie process in memory - another copy of *INT386W.SYS*. Terminate any zombie processes from the task-manager, taking care not to terminate the working client prolog process!

This predicate halts a Prolog server of the given Id:

```
halt_prolog( Id ) :-
  winapi( (int386w,'HaltProlog'), [Id], 0, _ ).
```

This predicate initialises a goal for execution by the Prolog server and returns whether this succeeded or not:

```
init_goal( Id, Goal, Output ) :-
  goal_string( Goal, Goalstring ),
  winapi( (int386w,'InitGoal'), [Id,Goalstring], 0,  Data ),
  get_result( Data, Output ).
```

This predicate calls a goal that has been previously initialised by init_goal/3:

```
call_goal( Id, Output ) :-
  winapi( (int386w,'CallGoal'), [Id], 0, Data ),
  get_result( Data, Output ) .
```

This predicate handles input from a goal which has been called on the Prolog server:

```
tell_goal( Id, Goal, Output ) :-
  goal_string( Goal, Goalstring ),
  winapi( (int386w,'TellGoal'), [Id,Goalstring], 0, Data ),
  get_result( Data, Output ) .
```

The following predicate exits a goal that has been initialised by init_goal/3:

```
exit_goal( Id ) :-
  winapi( (int386w,'ExitGoal'), [Id], 0,  _ ).
```

The following predicates are utilities to simplify the above.

```
get_result( Buffer, Result  ) :-
  (   Buffer > 0
  -> wintxt( Buffer, 0, 0, Result )
  ;   Result = Buffer
  ) .

goal_string(  Goal, Goalstring ) :-
  ( writeq(Goal),
    write(   .   )
  ) ~> Goalstring .
```

When these predicates have been compiled you can then start to test the Prolog server code. In this example we show the *CLIENT.PL* example being controlled from the ***WIN-PROLOG*** command-line. The first thing to do is to load the server copy of ***WIN-PROLOG*** :

```
?- load_prolog( Id ).

Id = 1
Yes
```

Then the example *CLIENT.PL* should be loaded. This is done by initialising the loading goal and then calling that goal:

```
?- init_goal( 1, ensure_loaded(prolog(client)), R ).
R = 'G 0000~M~J'

?- call_goal( 1, R ).
R = 'T 0000~M~J# 0.09 seconds to consult client.pl  [c:\tempis\] ~M~J'
```

Once the loading goal has finished it can be removed from the call stack:

```
?- exit_goal( 1 ).
Yes
```

Now the *CLIENT.PL* program is loaded into the Prolog server, we can initialise and run the *menu/0* program that kicks off the meals example:

```
?- init_goal( 1, menu, Output ).
Output = 'G 0000~M~J'

?- call_goal( 1, R ).
R = 'I 0000~M~JDo you want meals of type:'
```

When this goal is run the initial result is a request for more input. We can provide this input using *tell_goal/2*:

```
?- tell_goal( 1, diet_meal, Output ).
Output = 'T 0000~M~J~M~JToday's delicious choice of diet_meal
menus~M~J~M~JStarter:~I~Iparma_ham_with_melon~M~J~I-oOo-~M~JMain
Course:~Idover_sole~M~J~I-oOo-~M~JDessert:~I~Ifresh_fruit_salad~M~J~I-oOo-
```

```
~M~JWine:~I~Imuscadet_sur_lie~M~J~M~Jprice:~I&24.43~Icalories:~I
625~M~J‘
```

This returns us our first solution. We can backtrack into the next solution by calling call_goal/2 again:

```
?- call_goal( 1, R ).
R = ‘T 0000~M~J~M~JToday’s delicious choice of diet_meal
menus~M~J~M~JStarter:~I~Iparma_ham_with_melon~M~J~I-oOo-~M~JMain
Course:~Ipoached_salmon~M~J~I-oOo-~M~JDessert:~I~Ifresh_fruit_salad~M~J~I-
oOo-~M~JWine:~I~Ichablis~M~J~M~Jprice:~I&29.60~Icalories:~I  625~M~J‘
```

The call_goal/2 predicate can be repeatedly called, to return more solutions. Eventually call_goal/2 will return a string such as:

```
R = ‘F 0001~M~J‘
```

This indicates there are no more solutions, and we can terminate the call session by:

```
?- exit_goal( 1, R ) .
R = 1
```

Finally when we’ve done checking that the program is working correctly we can then shut down this Prolog server:

```
?- halt_prolog( 1 ).
Yes
```

Note that this does not shut down the DLL, so you can open a new server by:

```
?- load_prolog( Id ).
```

Chapter 11 The Intelligence Server interface to *flex*

Introduction

This part of the *Intelligence Server* toolkit requires that you have purchased the *flex* expert system toolkit.

Flex is an expert system and as an expert system it sometimes requires input from the user. The input from the user is provided through pre-defined dialogs. A single choice selection dialog for selecting one item from many. A multi-choice selection dialog for selecting many items from many items. An edit dialog to allow the user to enter text, and a message box that can provide yes no or ok cancel type questions. In each of the programming languages used in the example below these dialogs are pre-defined, though of course the developer can redefine them, within the constraints of the interface.

General Description

The *Intelligence Server* interface to *flex* is analagous to the *Intelligence Server*; a *flex* object is defined that has the following functions defined:

```
LoadFlex
HaltFlex
InitGoal
CallGoal
ExitGoal
RunGoal
```

Unlike the *Intellgence Server* the details of managing the ID of the prolog instance is encapsulated in the *flex* object. So in Visual Basic 6 for example you can call one of the above functions in a *flex* object, flx, say as:

```
flx.LoadFlex(     )
```

or

```
flx.InitGoal( "reconsult_rules(  robbie.ksl  ). " )  Note the full-stop followed by a space
flx.CallGoal
flx.ExitGoal
```

So there is no reference to the ID of a prolog instance, that instance is instead represented by the object, although you can obtain that instance if needed.

The last function, RunGoal, does all the work. With this you call your start program for your *flex* application, for example in Delphi:

```
Text1.text := flx.RunGoal( "run. " );
```

Which will execute the goal run/0 and return its results to Text1.text. Internally RunGoal handles the *flex* calls to the pre-defined dialogs as and when required.

Particulars

The VB *flex* example resides in the directory <your prolog>\SERVER\EXAMPLES\VB\FLEX, copy the files there to a new directory. The files FLEXSERV.PL, INT386W.BAS, INT386W.DLL, INT386W.OVL and VINT386W.DLL should be copied to this directory from the <your prolog>\SERVER\INCLUDES directory. FLEX.PC should be copied from the <your prolog>\SYSTEM directory, and the ROBBIE.KSL file copied from the <your prolog>\EXAMPLES\flex directory. Lastly the prolog engine INT386W.SYS should be copied from the <your prolog> directory.

Example Code

Visual Basic .NET

The Visual Basic .NET example comprises the files:

- FLEX.VB - The flex class definition
- LPA.DLL - The .NET interface to the INT386W.DLL
  Note, this can also be included as source code defined in the file INTSERVER.VB. You will find this file in the SERVER/INCLUDES/MS.NET directory.
- FORM1.VB* - The main form.
- LSTBOX.VB* - Single choice selection dialog
- MLTBOX.VB* - Multiple choice selection dialog
- EDTBOX.VB* - Edit box dialog.
- FLXUTIL.VB* - A helper module for the dialogs.
- ROBBIE.KSL - The flex code we will consult, using reconsult_rules/1 .
- FLEX.PC - The flex runtime code
- INT386W.SYS - The Prolog engine.
- INT386W.OVL - The Intelligence Server runtime
- INT386W.DLL - The Intelligence Server DLL

To build an application you will need to include all these files in your project, the example files live in the SERVER/EXAMPLES/MS.NET/vbFlex directory, those marked above with an asterisk (*). The easiest way to set this up is to copy the files in the example directory into a new directory and open the project file **FLEXEG.VBPROJ** from within visual studio. The other files are copied from the appropriate places in the includes directory, these files should be set to Copy Always in their Visual Studio properties, that way they will be copied into the build directory on compile and will allow the application to run within the Visual Studio IDE.

The code from the file **FORM1.VB,** this is the main form, which gets displayed when the application starts.

```
Public Class Form1
```

Create a new flex instance.

```
    Dim flx As New flex("")
```

Define line end string.

```
    Const crlf As String = Chr(13) + Chr(10)
```

On loading the form use *reconsult_rules/1* to consult the ROBBIE.KSL flex file, and set the text box to the result of that call. Note that flex and prolog are already loaded, this is happens when the flex instance is created.

```
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

        flx.initGoal("reconsult_rules( 'robbie.ksl' ). ")

        TextBox1.Text = flx.callGoal()

        flx.exitgoal()

    End Sub
```

On clicking Button1 (The help button) display a simple help message.

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click

        MsgBox("This program demonstrates the use of flex with the Intelligence Server,"
+ crlf + " in this case the example program ROBBIE.KSL." + crlf + "Press the Run
button to run robbie!", MsgBoxStyle.OkOnly, "Flexeg")

    End Sub
```

On clicking Button2 (The run button) execute the flex start goal, in this case the goal *run/0* . Not ice the space following the fullstop in the goal, this is always required, and will cause an error if missing.

```
    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click

        TextBox1.Text = ""

        TextBox1.Text = flx.rungoal("run. ")

    End Sub

End Class
```

The code from the file **LSTBOX.VB**, we will only examine one of the dialog files, as the others are very similar. **LSTBOX.VB** defines a single choice menu dialog.

```
Public Class lstbox

    Private result As String
```

This is the main entry point to the dialog, a set of items is passed into the lstbox as a string, that is parsed and then put into the list box control for the user to choose from.

```
    Public Function listbox(ByVal instring As String) As String
```

```
        Dim remainder As String = ""

        Dim item As String
```

Ensure that the listbox control is empty.

```
        ListBox1.Items.Clear()
```

Set the title of the dialog to the first item in the string.

```
        Me.Text = flxutil.getitem(instring, remainder)
```

Set the prompt of the dialog to the next item in the string.

```
        TextBox1.Text = flxutil.getitem(remainder, remainder)
```

While there are more items, add them to the listbox.

```
        While remainder <> ""

          item = flxutil.getitem(remainder, remainder)

          ListBox1.Items.Add(item)

        End While

        result = ""
```

Show the dialog as a modal dialog.

```
        Me.ShowDialog()
```

And when finished assign the result to the function.

```
        listbox = result

    End Function
```

On clicking the OK button, the selected Item is bound to the result variable and the dialog is hidden.

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles Button1.Click

        result = "'" + ListBox1.SelectedItem + "' "

        Me.Hide()

    End Sub
```

On clicking the explain button, the result is bound to the special string '??' that indicates to flex that the user has requested an explanation.

```
    Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles Button2.Click

        result = "??"
```

```
        Me.Hide()

    End Sub


    Private Sub lstbox_FormClosed(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosedEventArgs) Handles MyBase.FormClosed

        Button2_Click(sender, e)

    End Sub

End Class
```

Visual Basic 6

The Visual Basic 6 example comprises the files:

- FLEX.CLS - The *flex* class definition.
- UTILITY.BAS - Some utilities.
- INT386W.BAS - The *Intelligence Server* definitions.
- MAIN.FRM - The main form.
- MLTBOX.FRM - Multiple choice selection dialog.
- LSTBOX.FRM - Single choice selection dialog.
- EDTBOX.FRM - Edit box dialog.
- ROBBIE.KSL - The flex code we are consulting
- FLEX.PC - The flex runtime code
- INT386W.SYS - The Prolog engine
- INT386W.OVL - The Intelligence Server runtime
- INT386W.DLL - The Intelligence Server DLL
- VINT386W.DLL - The Visual Basic 6 interface to the Intelligence Server DLL

To build an application you will need all these files in one directory. The dialog forms correspond to the dialogs in the prolog-based environment for *flex*.

The code from the file MAIN.FRM.

```
'This is a Visual Basic 6 example program demonstrating the Intelligence Server
'version of flex using the example program "ROBBIE.KSL"
```

Create a new flex instance.

```
Dim flx As New flex
```

This sub-routine runs the flex goal *run*.

```
Private Sub Command1_Click()
  Text1 = ""
  Text1 = flx.RunGoal("run . ")
End Sub
```

On loading the form flex is loaded through the following subroutine.

```
Private Sub Form_Load()
  flx.LoadFlex ("")
  flx.InitGoal ("reconsult_rules( 'flex\robbie.ksl') ). ")
  flx.CallGoal
  flx.ExitGoal
End Sub
```

On closing the form flex is halted.

```
Private Sub Form_Unload(Cancel As Integer)
  flx.HaltFlex
End Sub
```

This sub-routine provides a simple help dialog.

```
Private Sub help_Click()
MsgBox "This program demonstrates the use of flex with the Intelligence Server," +
Chr(13) + " in this case the example program ROBBIE.KSL." +  Chr(13) + "Press the
Run button to run robbie!", vbOKOnly, "Flexeg"
End Sub
```

Delphi

The Delphi *flex* example resides in the directory <your prolog>\SERVER\EXAMPLES\DELPHI\FLEX, copy the files there to a new directory. The files FLEXSERV.PL, INTELLIGENCESERVER.PAS, INT386W.DLL, INT386W.OVL should be copied to this directory from the <your prolog>\SERVER\INCLUDES directory. FLEX.PC should be copied from the <your prolog>\SYSTEM directory, and the ROBBIE.KSL file copied from the <your prolog>\EXAMPLES\flex directory. Lastly the prolog engine INT386W.SYS should be copied from the <your prolog> directory.

The Delphi example comprises the files:

- MAIN.PAS - The main form.
- FLEX.PAS - The *flex* class definition.
- FLEXUTIL.PAS - Some utilities.
- INTELLIGENCESERVER.PAS - The *Intelligence Server* definitions.
- MULTI.PAS - Multiple choice selection dialog.
- SINGLE.PAS - Single choice selection dialog.
- EDIT.PAS - Edit box dialog.
- ROBBIE.KSL - The flex code we are consulting

- FLEX.PC - The flex runtime code
- INT386W.SYS - The Prolog engine
- INT386W.OVL - The Intelligence Server runtime
- INT386W.DLL - The Intelligence Server DLL

To build an application you will need all these files. The dialog forms correspond to the dialogs in the prolog-based environment for *flex*.

The code from the file **MAIN.PAS**.

```
unit main;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls,flex;

type
  TForm1 = class(TForm)
    Text1: TMemo;
    label1: TLabel;
    Run: TButton;
    Help: TButton;
    procedure FormCreate(Sender: TObject);
    procedure RunClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure HelpClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  flx:  TFlex;

implementation

{$R *.DFM}
```

When the run button is clicked the goal "run" is executed, notice the white space following the full stop.

```
procedure TForm1.RunClick(Sender: TObject);
begin
   Text1.Text := "";
   Text1.Text := flx.RunGoal( "run. " );
end;
```

When the form is created flex is loaded.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   flx.LoadFlex("");
   flx.InitGoal( "reconsult_rules('robbie.ksl') ). " );
   flx.CallGoal;
   flx.ExitGoal;
end;
```

On destroying the form flex is halted.

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  flx.HaltFlex;
end;
```

This procedure provides a simple help dialog.

```
procedure TForm1.HelpClick(Sender: TObject);
begin
  MessageDlg( 'This program demonstrates the use of flex with the Intelligence Server,
in this case the example program ROBBIE.KSL. Press the Run button to run robbie!',
mtCustom, [mbOk], 0 );
end;

end.
```

Java

The Java *flex* example resides in the directory <your prolog>\SERVER\EXAMPLES\JAVA\FLEX, copy the files there to a new directory. The files FLEXSERV.PL, Int386w.java, INT386W.DLL, INT386W.OVL and JINT386W.DLL should be copied to this directory from the <your prolog>\SERVER\INCLUDES and <your prolog>\SERVER\INCLUDES\LPA\IS directory. FLEX.PC should be copied from the <your prolog>\SYSTEM directory, and the ROBBIE.KSL file copied from the <your prolog>\EXAMPLES\flex directory. Lastly the prolog engine INT386W.SYS should be copied from the <your prolog> directory.

The Java example comprises the files:

- FlexEG.JAVA - The main form.
- FlexIS.JAVA - The *flex* class definition.
- Int386w.JAVA - The *Intelligence Server* definitions.
- MltBox.JAVA - Multiple choice selection dialog.
- LstBox.JAVA - Single choice selection dialog.
- EdtBox.JAVA - Edit box dialog.
- ROBBIE.KSL - The flex code we are consulting
- FLEX.PC - The flex runtime code
- INT386W.SYS - The Prolog engine
- INT386W.OVL - The Intelligence Server runtime

- INT386W.DLL - The Intelligence Server DLL
- JINT386W.DLL - The Java DLL interface to the Intelligence server DLL

To build an application you will need all these files. The dialog forms correspond to the dialogs in the prolog-based environment for *flex*.

The **FlexEG.JAVA** code

```
// Main entry point into the Java ROBBIE.KSL

import java.awt.*;
import java.awt.event.*;
import FlexIS;
import MsgBox;

public class FlexEG extends Frame
{
  Button   runButton, helpButton;
  TextArea outputText;
  FlexIS   flex;
  static Frame    thisFrame;

  public static void main( String[] main_args )
  {
    thisFrame = new FlexEG();
  }

  // Constructor
  public FlexEG()
  {
    super( "Robbie the Robot" );

    // Create a new flex object
    flex = new FlexIS(this);

    // Load flex
    flex.loadFlex("");

    GridLayout grid = new GridLayout(3,1);

    Panel buttonPanel = new Panel( grid );
    Panel textPanel   = new Panel(new GridLayout(1,1));

    this.add( textPanel,   BorderLayout.CENTER );
    this.add( buttonPanel, BorderLayout.EAST );

    runButton  = new Button( "Run" );
    helpButton = new Button( "Help" );

    outputText = new TextArea();
    outputText.setEditable( false );

    buttonPanel.add( runButton );
```

```
    buttonPanel.add( helpButton );

    textPanel.add( outputText );

    // Load the flex KSL code
    flex.initGoal(   "reconsult_rules( robbie.ksl ). " ;
    outputText.setText( flex.callGoal() );
    flex.exitGoal();

    this.addWindowListener( new WindowAdapter()
                {
                  public void       windowClosing(WindowEvent e)
                  {
                    flex.haltProlog();
                    System.exit(0);
                  }
                }
               );


    runButton.addActionListener( new  ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
outputText.setText( "" );
          outputText.setText( flex.runGoal( "run. " ) );
        }
      }
    );

    helpButton.addActionListener( new ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
          MsgBox Msg = new MsgBox( thisFrame );
          Msg.MessageBox( ":ok:This program demonstrates the use
of flex with the Intelligence Server, in this case the example program ROBBIE.KSL.
Press the Run button to run Robbie!" );
        }
      }
    );

    this.pack();
    this.show();
  }
}
```

Chapter 12 Creating a COM object in Visual Basic 6

Visual Basic 6 provides a very easy way to turn the Intelligence Server DLL into a COM DLL that can be accessed from any language that supports COM. To achieve that you will need to create a wrapper DLL around the Intelligence Server interface as follows.

Create a new project of type ActiveX DLL.

Rename the class as IS say (or whatever you choose)

Save the class file created as LPA.CLS

Add the module INT386W.BAS from the folder <your prolog>\SERVER\INCLUDES

Select the Class section of the code editor for the LPA.CLS file

Add a private variable ID as long

Add a new public function LoadProlog setting its return type as Long

```
Public Function LoadProlog(CommandLine As String,BufferSize as Integer,  Encoding
As Integer, Tickle As Integer) As Long
```

In the code window type, within the function type:

```
PrologId = Prolog.LoadProlog(CommandLine,BufferSize,Encoding)

LoadProlog = PrologId
```

Add similar class functions for each of the Prolog functions and Sub-Routines. You will hopefully end up with something like the following:

```
Private PrologId As Long

Public Function LoadProlog(CommandLine As String,BufferSize as Integer,  Encoding
as Integer, Tickle as Integer) As Long
    PrologId = Prolog.LoadProlog(CommandLine,BufferSize,Encoding,Tickle)
  LoadProlog = PrologId
End Function

Public Sub HaltProlog()
  Prolog.HaltProlog PrologId
End Sub

Public Function InitGoal(Goal As String) As String
  InitGoal = Prolog.InitGoal(PrologId, Goal)
End Function

Public Function CallGoal() As String
  CallGoal = Prolog.CallGoal(PrologId)
End Function
```

```
Public Sub ExitGoal()
  Prolog.ExitGoal PrologId
End Sub

Public Function TellGoal(Goal As String) As String
  TellGoal = Prolog.TellGoal(PrologId, Goal)
End Function

Public Property Get ID() As Variant
   ID = PrologId
End Property
```

Having done that check the class attributes are as follows:

```
DataBindingBehaviour   = 0 - vbNone
DataSourceBehaviour    = 0 - vbNone
Instancing             = 5 - Multiuse
MTSTransactionMode     = 0 - NotAnMTSObject
Persistable            = 0 - NotPersistable
```

Save the project as LPAIS.VBP and save the class file again.

Finally build the DLL.

To use the DLL you need to ensure that INT386W.SYS is visible to the COM DLL by being on the path and that the VINT386W.DLL, INT386W.DLL and INT386W.OVL are in the same directory as INT386W.SYS. You can now use the COM DLL just like any other.

Chapter 13 Efficient Data Management

It is often the case that you need to pass large data structures or many small pieces of data to and from ***WIN-PROLOG*** . This can be a tiresome and inefficient exercise if each piece of data is sent by a call to the Intelligence Server, or the C program has to unravel a long and complicated ***WIN-PROLOG*** structure. To demonstrate a technique that is much more efficient and that can be generally applied, an example is developed based on the game Reversi.

The purpose of this example is to show how to pass large data structures to and from a front-end defined in C or C++ and ***WIN-PROLOG*** represented by the Intelligence Server. It is not intended to show how to write a good Reversi program! The aim here is simplicity, the reversi board is implemented in a shared data area that is created as a so-called "Memory Mapped File" (MMF). A Windows MMF is a data area that allows two or more different processes to share memory. Normally this is not possible and processes are well protected from each other. The MMF maps a data area into each process s address space and behaves almost like any other area of allocated memory except that it is shared.

How it works.

The example works in the following way. ***WIN-PROLOG*** s job is to know about the game and to manipulate the pieces on the board according to the rules of the game. C's job is to represent the pieces visually on the screen, and to manage the user interaction with the program. Firstly an MMF area is created with a given name in ***WIN-PROLOG*** and mapped into ***WIN-PROLOG*** 's address space. Then the same MMF area is created in C with the given name and mapped into C's address space. The two programs now have a shared area of memory that represents the board. The interaction between C and ***WIN-PROLOG*** behaves as follows. The C program receives a relevent event, in response to that event it can call ***WIN-PROLOG*** with one of three things, create a new board, make a human move, or make a computer move. Then in response to ***WIN-PROLOG*** 's answer it redraws the board from the MMF and either displays an end of game message or waits for another event. That's it, the entire game from C's point of view!

The Files

The C reversi example resides in the directory <your prolog>\SERVER\EXAMPLES\C\REVERSI, copy the files there to a new directory. The files REVERSI2.PL, INT386W.H, INT386W.DLL and INT386W.OVL should be copied to this directory from the <your prolog>\SERVER\INCLUDES directory. Lastly the prolog engine INT386W.SYS should be copied from the <your prolog> directory.

The C code.

The main application file loads ***WIN-PROLOG*** , consults "REVERSI2.PL" and calls the main dialog. When the main dialog finishes, ***WIN-PROLOG*** is halted and the application ends.

Declare the functions and data items used in the program.

```
/*****************************************************************
 * Reversi Example for WIN-PROLOG - by Alan Westwood - 27 Jun 01      *
 * Copyright (c) 2001 - Logic Programming Associates Ltd             *
 *****************************************************************/

#include <windows.h>
#include "resource.h"
```

```
#include  int386w.h

int     WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int) ;
VOID    WINAPI CallOneGoal(UINT,LPSTR) ;
BOOL    WINAPI ReversiProc(HWND,UINT,WPARAM,LPARAM) ;
BOOL    WINAPI BoardProc(HWND,UINT,WPARAM,LPARAM) ;
BOOL    WINAPI NewBoard(UINT,UINT) ;
void    WINAPI CloseBoard() ;
void    WINAPI DrawAll(HWND,HDC) ;
void    WINAPI DrawPieces(HDC) ;
void    WINAPI SetRadio(HWND,UINT) ;
UINT    WINAPI GetRadio(HWND) ;

UINT        Prolog ;
HANDLE      Instance ;
HWND        ReversiWindow ;
HWND        BoardWindow ;
UCHAR       String[4096] ;
```

The board is represented by an array of LONG; the variable Board is a pointer to that array. The MapHandle is a handle to the MMF.

```
LPLONG      Board ;
UINT        BoardSize ;
HANDLE      MapHandle ;

RECT        ClientRect ;
UINT        sqHeight ;
UINT        sqWidth ;
```

These are various GDI objects used in the display of the game.

```
HGDIOBJ     BlackBrush ;
HGDIOBJ     WhiteBrush ;
HGDIOBJ     GrayBrush ;
HBRUSH      WindowBrush ;
HCURSOR     CrossCursor ;
HCURSOR     NormalCursor ;
```

A macro that performs the arithmetic required to access a two dimensional array in a linear address space.

```
#define ARRAY(x,y) *(Board+(((y)*BoardSize + (x))))
#define CAPTION "Reversi"

enum pieces { White=-1, Empty, Black } ;

// main procedure loads Prolog and an example program and executes a dialog

int WINAPI  WinMain(
   HINSTANCE    hInstance,
   HINSTANCE    hPrevInstance,
   LPSTR        lpCmdLine,
```

```
    int             nCmdShow
    )
{
    Instance = hInstance ;

    Prolog = LoadProlog("",0,0,0) ;

    if  (Prolog < 0)
    {
        wsprintf(String,"LoadProlog Error %d!",Prolog) ;
        MessageBox(NULL,String,CAPTION,MB_OK|MB_ICONHAND) ;
        return(FALSE) ;
    }

    CallOneGoal(Prolog,"ensure_loaded(prolog(reversi2)). ") ;
```

Try to create a board, if that fails then show a relevant message and quit.

```
    if  (!NewBoard(Prolog,8))
    {
        MessageBox(NULL,
                "Board creation failed",
                CAPTION,
                MB_OK|MB_ICONHAND) ;
        HaltProlog(Prolog) ;
        return(FALSE) ;
    } ;
```

Initialize GDI objects used in the program.

```
    BlackBrush    = GetStockObject(BLACK_BRUSH ) ;
    WhiteBrush    = GetStockObject(WHITE_BRUSH ) ;
    CrossCursor   = LoadCursor(NULL,IDC_CROSS) ;
    NormalCursor  = LoadCursor(NULL,IDC_ARROW) ;
```

Call the dialog, when the dialog is finished, close the board and halt *WIN-PROLOG* .

```
    DialogBox(Instance,
            MAKEINTRESOURCE(IDD_REVERSI_DIALOG),
            NULL,
            ReversiProc) ;
    CloseBoard() ;
    HaltProlog(Prolog) ;
    return(TRUE) ;
}

// call a goal once and once only, and put the result into the buffer String

VOID  WINAPI    CallOneGoal(UINT Prolog, LPSTR Goal)
{
    InitGoal(Prolog,Goal) ;
    lstrcpy(String,CallGoal(Prolog)) ;
    ExitGoal(Prolog) ;
```

```
}

// handle messages for the reversi dialog

BOOL  WINAPI    ReversiProc(
   HWND    Window,
   UINT    Message,
   WPARAM wParam,
   LPARAM  lParam
   )
{
   HDC           dc ;
   PAINTSTRUCT  paint ;
   UINT          Row ;
   UINT          Col ;
   int           Result;

   switch    (Message)
   {
```

The default cursor for the window is set to 0 using SetClassLong(Window, GCL_HCURSOR, 0), because we are changing the cursor ourself if we didn't do this the cusor would revert to a pointer every time the mouse moved.

```
      caseWM_INITDIALOG :
          SetClassLong( Window, GCL_HCURSOR, 0 ) ;
          ReversiWindow = Window ;
          break ;

      caseWM_COMMAND :
          switch     (wParam)
          {
```

When the "New Game" button is clicked the "Board Size" dialog is run. The result from that is then used to create a new board of that size. If successful the window is then redrawn, otherwise a message is displayed and the dialog is quit.

```
          case  IDC_NEWGAME :
                Result = DialogBox(Instance,
                           MAKEINTRESOURCE(IDD_BOARDSIZE),
                           Window,
                           BoardProc) ;
                if    ( Result==TRUE )
                {
                      if    (NewBoard(Prolog,BoardSize))
                      {
                            dc = GetDC(Window) ;
                            DrawAll(Window,dc) ;
                            ReleaseDC(Window,dc) ;
                      }
                      else
                      {
```

```
                    MessageBox(Window,
                              "Board creation failed",
                              CAPTION,
                              MB_OK|MB_ICONHAND) ;
                    EndDialog(Window,TRUE) ;
                    return(TRUE) ;
               }
          }
          return(TRUE) ;
          break ;
```

When the "Pass" button is clicked it is the computer's turn to move so the goal "computer. " is called. The pieces are then redrawn. If the string returned from the call to ***WIN-PROLOG*** is not "ok" the game has finished and that string is shown in a message box.

```
     case  IDC_PASS :
          CallOneGoal(Prolog,"computer. ") ;
          dc = GetDC(Window) ;
          DrawPieces(dc) ;
          ReleaseDC(Window,dc) ;

          if     (strcmp(String+8,"ok"))
          {
               MessageBox(Window,String+8,CAPTION,MB_OK) ;
          }

          return(TRUE) ;
          break ;

     }

     return(TRUE) ;
     break ;
```

When the left mouse button is released on the dialog, the array indexes Col and Row are calculated from the mouse position. The Col and Row are then checked for validity. If the Col and Row are valid, then these are passed as parameters to the ***WIN-PROLOG*** goal "human(Col,Row). " to play the human's move. If that call has not failed i.e. the result is not 'F' then the pieces are redrawn. If the text returned from the call is not equal to "ok" then the game has ended, and so a message box is shown with that text. Otherwise ***WIN-PROLOG*** is called again with "computer. " to play the computer's move. The pieces are then drawn again. If the text from the computer's move is not "ok" then the game has ended so that text is shown in a message box. Finally the cursor is reset to the NormalCursor.

```
caseWM_LBUTTONUP :
     Col = LOWORD(lParam) / sqWidth ;
     Row = BoardSize - (HIWORD(lParam) / sqHeight) - 1 ;

     if     (      Row >= 0 &&
                   Row < BoardSize &&
                   Col >= 0 &&
                   Col < BoardSize
          )
     {
```

```
                wsprintf(String,"human(%d,%d). ", Col, Row ) ;
                CallOneGoal(Prolog,String) ;

                if      (String[0] != 'F')
                {
                        dc = GetDC(Window) ;
                        DrawPieces(dc) ;
                        ReleaseDC(Window,dc) ;

                        if      (strcmp(String+8,"ok"))
                        {
                                MessageBox(Window,
                                        String+8,
                                        CAPTION,
                                        MB_OK) ;
                        }
                        else
                        {
                                CallOneGoal(Prolog, "computer. ") ;
                                dc = GetDC(Window) ;
                                DrawPieces(dc) ;
                                ReleaseDC(Window,dc) ;

                                if      (strcmp(String+8,"ok"))
                                {
                                        MessageBox(Window,
                                                String+8,
                                                CAPTION,
                                                MB_OK) ;
                                }
                        }
                }
        }

        SetCursor(NormalCursor) ;
        break ;
```

On a mouse move the array indexes Col and Row are calculated from the mouse position. The Col and Row are then checked for validity. If the Col and Row are valid then the ***WIN-PROLOG*** goal "valid(Col,Row). " is called. If that goal fails by returning 'F' as the result then the cursor is set to NormalCursor, otherwise the cursor is set to CrossCursor. Thus the cross cursor indicates squares that the human can play on the board.

```
caseWM_MOUSEMOVE :
        Col = LOWORD(lParam) / sqWidth ;
        Row = BoardSize - (HIWORD(lParam) / sqHeight) - 1 ;
        wsprintf(String,"valid(%d,%d). ", Col, Row) ;
        CallOneGoal(Prolog,String) ;

        if      (String[0] == 'F')
        {
                SetCursor(NormalCursor) ;
        }
```

```
            else
            {
                  SetCursor(CrossCursor) ;
            }
            break ;
```

On a paint message the window is redrawn.

```
      caseWM_PAINT :
            dc = BeginPaint(Window,&paint) ;
            DrawAll(Window,dc) ;
            EndPaint(Window,&paint) ;
            return(TRUE) ;
            break ;

      caseWM_CLOSE :
            EndDialog(Window,TRUE) ;
            return(TRUE) ;
            break ;

      caseWM_DESTROY :
            ReversiWindow = NULL ;
            break ;
      }
      return(FALSE) ;
}
```

The Board modal dialog gets and sets the BoardSize variable.

```
// handle messages for the Board dialog

BOOL  WINAPI    BoardProc(
      HWND    Window,
      UINT    Message,
      WPARAM wParam,
      LPARAM lParam
      )
{
      BOOL    Result ;

      switch    (Message)
      {
      caseWM_INITDIALOG :
            BoardWindow = Window ;
            SetRadio(Window,BoardSize) ;
            break ;

      caseWM_COMMAND :
            switch      (wParam)
            {
            case  IDOK :
                  BoardSize = GetRadio(Window) ;
```

```
                Result = TRUE ;
                break ;
            case  IDCANCEL :
                Result = FALSE ;
                break ;
            default :
                return(FALSE) ;
            }

            EndDialog(Window,Result) ;
            return(TRUE) ;
            break ;

        caseWM_CLOSE :
            return(TRUE) ;
            break ;

        caseWM_DESTROY :
            BoardWindow = NULL ;
            break ;
        }
        return(FALSE) ;
    }
```

DrawAll redraws the entire board, on a paint message or in the event of a new board created. The height of the client rectangle of the window is used as the width of the board; this ensures a square board. The background for the board is painted in the system colour COLOR_BTNFACE and the brush for that is created here. This ensures that if the system colours change then the board colour will change accordingly. The pieces are then drawn.

```
    // Draw the board into the device context

    void    WINAPI     DrawAll( HWND Window, HDC dc)
    {
        WindowBrush = CreateSolidBrush((COLORREF)GetSysColor(COLOR_BTNFACE)) ;

        GetClientRect(Window,&ClientRect) ;
        sqHeight = (ClientRect.bottom-ClientRect.top) / BoardSize ;
        ClientRect.right = ClientRect.bottom ;

        sqWidth = sqHeight ;

        FillRect(dc,&ClientRect,WindowBrush) ;

        DeleteObject( WindowBrush ) ;
        DrawPieces(dc) ;
    }
```

DrawPieces draws all the pieces by iterating through the array. Empty "squares" are shown as ellipses drawn in the system colour COLOR_BTNFACE and the brush for that is created here. This ensures that if the system colours change then the board colour will change accordingly.

```
// Draw the pieces into the device context

void    WINAPI    DrawPieces( HDC Dc )
{
   UINT     Row ;
   UINT     Col ;
   int Piece ;
   int DcSaved;

   WindowBrush = CreateSolidBrush((COLORREF)GetSysColor(COLOR_BTNFACE)) ;

   DcSaved = SaveDC( Dc );

   for (Col=0; Col<BoardSize; Col++)
   {
       for (Row=0; Row<BoardSize; Row++)
       {
            Piece = (ARRAY(Col,(BoardSize-1-Row))) ;

            if     (Piece == Empty )
            {
                 SelectObject(Dc,WindowBrush ) ;
            }
            else if (Piece == Black )
            {
                 SelectObject(Dc,BlackBrush) ;
            }
            else if (Piece == White )
            {
                 SelectObject(Dc,WhiteBrush) ;
            }

            Ellipse(Dc,
                 Col*sqWidth,
                 Row*sqHeight,
                 (Col+1)*sqWidth,
                 (Row+1)*sqHeight) ;
       }
   }

   RestoreDC( Dc, DcSaved );

   DeleteObject( WindowBrush ) ;
}
```

SetRadio sets the radio button states for the BoardSize dialog from the given size.

```
// Set the size radio button

void    WINAPI    SetRadio(HWND Window, UINT Size)
{
   int ID ;
```

```
    switch(Size)
    {
    case6:
        ID = IDC_TINY ;
        break ;
    case8:
        ID = IDC_SMALL ;
        break ;
    case10:
        ID = IDC_NORMAL ;
        break ;
    case12:
        ID = IDC_BIG ;
        break ;
    case14:
        ID = IDC_BIGGER ;
        break ;
    case16:
        ID = IDC_HUGE ;
        break ;
    default :
        ID = IDC_SMALL ;
        break ;
    }

    SendDlgItemMessage(Window,
              ID,
              BM_SETCHECK,
              (WPARAM) BST_CHECKED,
              (LPARAM) 0) ;
}

// Return the board size associated with the set radio button

UINT  WINAPI    GetRadio( HWND Window)
{
    if   (SendDlgItemMessage(Window,IDC_TINY,BM_GETCHECK,0,0)
              == BST_CHECKED)
    {
        return(6) ;
    }
    else if (SendDlgItemMessage(Window,IDC_SMALL,BM_GETCHECK,0,0)
              == BST_CHECKED)
    {
        return(8) ;
    }
    else if (SendDlgItemMessage(Window,IDC_NORMAL,BM_GETCHECK,0,0)
              == BST_CHECKED)
    {
        return(10) ;
    }
    else if (SendDlgItemMessage(Window,IDC_BIG,BM_GETCHECK,0,0)
```

```
            == BST_CHECKED)
    {
        return(12) ;
    }
    else if (SendDlgItemMessage(Window,IDC_BIGGER,BM_GETCHECK,0,0)
            == BST_CHECKED)
    {
        return(14) ;
    }
    else if (SendDlgItemMessage(Window,IDC_HUGE,BM_GETCHECK,0,0)
            == BST_CHECKED)
    {
        return(16) ;
    }
    else
    {
        return(8) ;
    }
}
```

The NewBoard function creates the board array from a MMF. First any open board is closed. Then ***WIN-PROLOG*** is called with the goal "new_board(Size). " to create a new board. If that call succeeds by returning the result 'T' then the text returned is the name of the file to create so CreateFileMapping is called with that name. The name is generated by ***WIN-PROLOG*** from the current time and date to avoid collisions. If the MMF was created succesfully then the Board is set to a view of the file by calling the function MapViewOfFile. If that was successful then the variable BoardSize is set to size and the function returns TRUE. If it was not successful then the MMF is closed and the function returns FALSE.

```
// Create a new board of the given size

BOOL  WINAPI    NewBoard(UINT prolog, UINT size)
{
    int ArraySize ;

    CloseBoard() ;
    wsprintf(String,"new_board(%d). ",size) ;
    CallOneGoal(prolog,String) ;

    if   (String[0] == 'T')
    {
        ArraySize = sizeof(LONG)*size*size ;
        MapHandle = CreateFileMapping((HANDLE)0xFFFFFFFF,
                                NULL,
                                PAGE_READWRITE,
                                0,
                                ArraySize,
                                String+8) ;

        if    (MapHandle != NULL)
        {
            Board = MapViewOfFile(MapHandle,
                                FILE_MAP_ALL_ACCESS,
```

```
                                0,
                                0,
                                ArraySize) ;

                if      (Board == NULL)
                {
                        CloseHandle(MapHandle) ;
                        MapHandle = NULL ;
                        return(FALSE) ;
                }
                else
                {
                        BoardSize = size ;
                        return(TRUE) ;
                }
        }
        else
        {
                return(FALSE) ;
        }

    }
    else
    {
        return(FALSE) ;
    }
}
```

The CloseBoard function closes an open MMF.

```
// Close a previously opened board

void    WINAPI  CloseBoard()
{
    if  (Board != NULL)
    {
        UnmapViewOfFile(Board) ;
    }

    if  (MapHandle != NULL)
    {
        CloseHandle(MapHandle) ;
    }

    BoardSize = 0 ;
}
```

The Prolog code

```
/* REVERSI II - Alan Westwood - LPA Ltd - 20 June 2001

 This version of Reversi is a command line driven program that uses a
```

```
memory buffer rather than assertions to keep the board state.

  The main predicates are:

  new_board/1 Create and initialize a new board with the given size

  computer/0  Tell Prolog to move if it can.
     Outputs one of the following
        'I won by n pieces'          - Prolog has won
        'You won by n pieces!'       - Prolog has lost
        'A draw'                     - It's a draw!
        'I pass'                     - Prolog cannot move
        ok                      - Still playing.

  human/2     Tell Prolog human moved at X,Y.
     Outputs one of the following
        'I won by n pieces'          - Prolog has won
        'You won by n pieces!'       - Prolog has lost
        'A draw'                     - It's a draw!
        'I pass'                     - Prolog cannot move
        ok                      - Still playing.
      or the predicate fails if not a valid move

  valid/2     Is this square a valid human move? This is used for hit
       testing.
       This will either succeed or fail, no output.

  Human pieces are represented by -1 in the array.
  Prolog pieces are represented by 1 in the array.
  Empty squares are zero.
  This makes it easy to check who has won.
*/
```

Prolog plays a move, the front end will call this. Try to make a computer move then check and report the game state. If it cannot make a move, then report that the computer passes.

```
% Prolog plays a move.

computer :-
  computer_move,
  !,
  state.
computer :-
  computer_pass .
```

The human makes a move. Try to make the move if that succeeds, check and report the game state. Otherwise fail.

```
% Human plays a move at X,Y

human(X,Y) :-
  human_move( X, Y ),
```

```
  state .

% Try and play a move by generate and test

computer_move :-
  computer( Who ),
  move( X, Y ),
  valid_move( X, Y, Who ),
  make_move( X, Y, Who ).

human_move( X, Y ) :-
  human( Who ),
  valid( X, Y )
  -> make_move( X, Y, Who ).
```

Valid human moves are cached after each computer move; this speeds up processing when the front end requests if a square is a valid move.

```
% Valid human move
% It's a valid move if it is in the cache.

valid( X, Y ) :-
  cache( Valid ),
  member( (X,Y), Valid ).
```

Report the state of the game. If there are no valid moves for either side then the game has ended. Count up the score. If the score is greater than zero then the computer has won. If the score is less than zero then the human has won. If the score is zero then it is a drawn game. Otherwise report "ok .

```
% Report the state of the game.

state :-
  (  no_more_moves
  -> sumall( Result ),
    (  Result > 0
    -> computer_won( Result )
    ;  Result < 0
    -> human_won( Result )
    ;  Result = 0
    -> draw
    )
  ;  ok
  ).

% Return the result of a finished game.

computer_won( Result ) :-
  write( 'I won by ' ),
  write( Result ),
  write( 'pieces!' ).
```

```
human_won( Result ) :-
  write( 'You won by ' ),
  R is abs(Result),
  write( R ),
  write( ' pieces!' ).

draw :-
  write( 'A draw!' ).

human_pass :-
  write( 'You must pass!' ).

computer_pass :-
  write( 'I pass' ).

ok :-
  write( ok ).

% It is a valid move if the square is empty and
% squares can be flipped.

valid_move( X, Y, Who ) :-
  empty_square( X, Y ),
  can_flip( X, Y, Who ).

% After making a move the cache is updated.

make_move( X, Y, Who ) :-
  put_board( X, Y, Who ),
  flip( X, Y, Who ),
  update_cache.

% To update the cache find all valid human moves.
% Caching valid moves avoids having to do this test
% every time the mouse moves.

update_cache :-
  findall( (X,Y),
         ( move(X,Y),
           valid_move(X,Y,-1)
         ),
         Valid
        ),
  retractall( cache(_) ),
  assert( cache(Valid) ).

% An empty square contains 0

empty_square( X, Y ) :-
  get_board( X, Y, 0 ).

% Generate a move
% The order of generation is the computer's simplistic strategy
```

```
move( X, Y ) :-
  board_size( Size ),
  ( corner( X, Y, Size )
  ; edge( X, Y, Size )
  ; inner_square( X, Y, Size )
  ; outer_square( X, Y, Size )
  ; outer_corner( X, Y, Size )
  ).

corner( X, Y, S ) :-
  S1 is S - 1,
  ( X = 0,   Y = 0
  ; X = 0,   Y = S1
  ; X = S1,  Y = S1
  ; X = S1,  Y = 0
  ).

edge( X, Y, S ) :-
  ( A = X,
    B = Y
  ; A = Y,
    B = X
  ),
  S1 is S - 3,
  integer_bound( 2, B, S1 ),
  ( A = 0
  ; A is S-1
  ).

inner_square( X, Y, S ) :-
  S1 is S - 3,
  integer_bound( 2, X, S1 ),
  integer_bound( 2, Y, S1 ).

outer_square( X, Y, S ) :-
  ( A = X,
    B = Y
  ; B = X,
    A = Y
  ),
  S1 is S - 3,
  integer_bound( 2, A, S1 ),
  ( B = 1
  ; B is S - 2
  ).

outer_corner( X, Y, S ) :-
  S1 is S - 1,
  S2 is S - 2,
  member( X, [0,1,S1,S2] ),
  member( Y, [0,1,S1,S2] ),
  \+ corner( X, Y, S ).
```

```
% Board predicates

new_board( Size ) :-
  repeat( 10 ),
  new_name( ArrayName ),
  create_array( ArrayName, Size ),
  !,
  retractall( board_size(_) ),
  assert( board_size(Size) ),
  write( ArrayName ),
  P1 is Size//2 - 1,
  P2 is P1 + 1,
  human( Human ),
  computer( Computer ),
  put_board( P1, P1, Human ),
  put_board( P1, P2, Computer ),
  put_board( P2, P1, Computer ),
  put_board( P2, P2, Human ),
  update_cache.

% Create a new name composed of "reversi" plus the current
% date and time (D,T)

new_name( Name ) :-
  time( 1, T ),
  ( write( reversi ),
    write( T )
  ) ~> Name.
```

Here the memory mapped file for the board is created. Notice that a mutex is grabbed, this stops other copies of the game from creating the same MMF whilst in the midst of creation. The MMF must be the first instance of this file created, or the predicate will fail. That is what the call to GetLastError checks. When all is done the mutex is released.

```
% Create an array given the arrayname, this is mutex protected.
% If this fails then the main application should close.

create_array( ArrayName, Size ) :-
  close_array( ArrayName ),
  ArraySize is 4 * Size * Size,
  write( ArrayName ) ~> ArrayStr,
  winapi( (kernel32,'CreateMutexA'),  [0,0,' MutexToProtectFileMapping' ], 0,
MutexHandle ),
  MutexHandle \= 0,
  winapi( (kernel32,'WaitForSingleObject'), [MutexHandle,5000], 0, WaitResult ),
  (  WaitResult = 0
  -> true
  ;  winapi( (kernel32,'ReleaseMutex' ), [MutexHandle], 0, _ ),
     fail
  ),
  % Got the mutex
```

```
  winapi( (kernel32,'CreateFileMappingA'), [16'FFFFFFFF,0,4,0,ArraySize,ArrayStr],
0, Handle ),
  Handle \= 0,
  winapi( (kernel32,'GetLastError'), [], 0, Error ),
  ( Error = 0
  -> winapi( (kernel32,'MapViewOfFile'), [Handle,16'f001f,0, 0, ArraySize], 0,
ArrayPtr ),
     ( ArrayPtr = 0
     -> winapi( (kernel32,'CloseHandle'), [Handle], 0, _ ),
       winapi( (kernel32,'ReleaseMutex' ), [MutexHandle], 0, _ ),
       fail
     ; winapi( (kernel32,'ReleaseMutex' ), [MutexHandle], 0, _ )
     )
  ; winapi( (kernel32,'CloseHandle'), [Handle], 0, _ ),
    winapi( (kernel32,'ReleaseMutex' ), [MutexHandle], 0, _ ),
    fail
  ),
  wintxt( ArrayPtr, ArraySize, 0, '' ),
  retractall( array_name(_,_,_) ),
  assert( array_name(ArrayName,ArrayPtr,Handle) ).

close_array( Name ) :-
  def( array_name, 3, _ ),
  array_name( Name, Pointer, Handle ),
  !,
  winapi( (kernel32,'UnmapViewOfFile'), [Pointer], 0, _ ),
  winapi( (kernel32,'CloseHandle' ),    [Handle], 0, _ ),
  retractall( array_name(_,_,_) ).
close_array( _ ).

% Get what's at X,Y
get_board( X, Y, What ) :-
  a_cell( X, Y, W ),
  What = W.

put_board( X, Y, What ) :-
  type( What, 1 ),
  a_cell( X, Y, What ).

% As we are dealing with a memory address rather than a
% Prolog buffer, some checks are made to make sure the X,Y
% position is a valid cell. If we did not do this there would
% be a risk of a GPF from invalid addresses.

a_cell( X, Y, What ) :-
  board_size( Size ),
  X >= 0,
  X < Size,
  Y >= 0,
  Y < Size,
  array_name( _, Array, _ ),
  Offset is 4*((Size * Y) + X),
  ( type( What, 0 )
```

```
  -> wintxt( (Array,Offset), 4, 0, String ),
     getx( 4, What ) <~ String,
     List = [What]
  ; putx( 4, What ) ~> String,
     wintxt( (Array,Offset), 4, 0, String )
  ).

% Flipping and testing

% Flip squares

flip( X, Y, Who ) :-
  to_flip( X, Y, Who, Flip ),
  flip_lines( Flip, Who ).

flip_lines( [], _ ).
flip_lines( [Line|Lines], Who ) :-
  ( Line = (X,Y)
  -> put_board( X, Y, Who )
  ; flip_lines( Line, Who )
  ),
  flip_lines( Lines, Who ).

% Find all squares to flip

to_flip( X, Y, Who, Flip ) :-
  findall( Line,
        get_squares( X, Y, Who, Line ),
        Flip
       ).

% Test to see if placing a piece at X,Y, pieces can be flipped.

can_flip( X, Y, Who ) :-
  get_squares( X, Y, Who, Line ),
  Line \= [],
  !.

get_squares( X, Y, Who, Squares ) :-
  opponent( Who, Opponent ),
  integer_bound( -1, DX, 1 ),
  integer_bound( -1, DY, 1 ),
  get_line( X, Y, DX, DY, (Who,Opponent), [], Squares ).

get_line( X, Y, DX, DY, (Who,Opponent), Found, Squares ) :-
  X1 is X + DX,
  Y1 is Y + DY,
  get_board( X1, Y1, Here ),
  ( Here   = Who
  -> Squares = Found
  ; Here   = Opponent
  -> get_line( X1, Y1, DX, DY, (Who,Opponent), [(X1,Y1)|Found], Squares)
  ).
```

```
opponent( -1,  1 ).
opponent(  1, -1 ).

human( -1 ).
computer( 1 ).
empty( 0 ).

no_more_moves :-
  \+
  (  board_size( S ),
     S1 is S - 1,
     ( human(Who)
     ; computer(Who)
     ),
     integer_bound( 0, X, S1 ),
     integer_bound( 0, Y, S1 ),
     valid_move( X, Y, Who )
  ).

% Sum the contents of the array

sumall( Sum ) :-
  board_size( S ),
  ArraySize is 4*S*S,
  array_name( _, Address, _ ),
  wintxt( Address, ArraySize, 0, Text ),
  sumall( 0, Sum ) <~ Text.

sumall( Count, Sum ) :-
  (  getx(4,X)
  -> NewCount is Count + X,
     sumall( NewCount, Sum )
  ;  Sum = Count
  ).
```

Appendix A: How to build and run the examples

Please note that the examples as supplied on the disk are not in a ready to run state. To run an example you need to do the following, please note that the set up for Java is slightly different please read the Java section, for Visual Studio 2008, and Visual Studio 2010 please read the Visual Studio section:

Assuming that your copy of Prolog is installed in its default directory, C:\PROGRAM FILES\WIN-PROLOG 4900\:

1. Create a new directory, C:\INTEG say.
2. Copy the contents of the particular example directory, including any sub-directories, into C:\INTEG.
3. Copy the generic files that are required by all *Intelligence Server* applications from C:\PROGRAM FILES\WIN-PROLOG 4900\SERVER\INCLUDES. These are:
    a. INT386W.DLL
    b. INT386W.OVL
    c. INT386W.SYS
4. Copy the language specific files from the appropriate language sub-directory of C:\PROGRAM FILES\WIN-PROLOG 4900\SERVER\INCLUDES directory. These are:
    a. C
        i. INT386W.LIB, from the main includes directory.
        ii. INT386W.H
    b. C++, which are in the C sub-directory.
        i. INT386W.CPP
        ii. INT386W.HPP
    c. Delphi
        i. INTELLIGENCESERVER.PAS
    d. VB6
        i. INT386W.BAS
        ii. VINT386W.DLL
    e. .NET
        i. LPA.DLL

ii. IntServer.VB
this is the source code for the LPA.DLL

f. Python

i. INT386W.PY

5. If using one of the flex examples copy the file FLEXSERV.PL from C:\PROGRAM FILES\WIN-PROLOG 4900\SERVER\INCLUDES\FLEX to C:\INTEG and the file FLEX.PC from C:\PROGRAM FILES\WIN-PROLOG 4900\SYSTEM to C:\INTEG.

6. If using one of the flex examples copy from C:\PROGRAM FILES\WIN-PROLOG 4900\SERVER\INCLUDES\FLEX to C:\INTEG the language specific files:

a. VB

i. FLEX.BAS

b. Delphi

i. FLEX.PAS

c. C++

i. FLEX.CPP

ii. FLEX.H

d. MS.NET

i. FLEX.VB

7. If using one of the flex examples copy the file FLEXSERV.PL from C:\PROGRAM FILES\WIN-PROLOG\4900\SERVER\INCLUDES\FLEX

8. Copy the example prolog code from C:\PROGRAM FILES\WIN-PROLOG 4900\SERVER\INCLUDES to C:\INTEG:

a. For the CLIENT examples copy the file CLIENT.PL

b. For the REVERSI examples copy the file REVERSI2.PL

c. For the ROBBIE flex examples copy the file C:\PROGRAM FILES\WIN-PROLOG 4900\EXAMPLES\FLEX\ROBBIE.KSL to C:\INTEG

9. Build the example depending on language from the C:\INTEG directory as :

a. C
run the batch file CLIENT.BAT

b. C++
Load and build the DSW file in the Visual Studio 6.0 environment.

c. Delphi
Load and build the DPR project file in the Delphi environment.

d. VB
   Load and build the VBP project file in the VB environment.

10. Run the built example by changing directory in a DOS box to C:\INTEG and:

a. C
   Run the .EXE file.

b. C++
   Copy the .EXE file from C:\INTEG\Release to the C:\INTEG directory and run it.

c. VB
   Run the .EXE file.

d. Delphi
   Run the .EXE file.

e. Python
   Run the appropriate .PY file.

Code Type	Files	Information
Generic code common to all	**INT386W.DLL**	**The Intelligence server interface.**
	INT386W.LIB	**Library definitions for the functions in INT386W.DLL**
	INT386W.OVL	**The Prolog component for the Intelligence Server interface.**
	INT386W.SYS	**The Prolog engine.**
Example Code	Client.PL	The Client example Prolog code.
	Reversi2.PL	The Reversi example Prolog code.
Flex Code	FlexServ.PL	The flex Prolog interface.
C	INT386W.H	Function declarations for the INT386W.DLL
C++	INT386W.HPP	The class declaration for the Int386w class.
	INT386W.CPP	The implementation code for the Int386w class.
	FLEX.CPP and FLEX.H	Flex interface file to the Intelligence Server.
VB6	INT386W.BAS	Function declarations for the VINT386W.DLL
	VINT386W.DLL	The Visual Basic 6 DLL interface to the INT386W.DLL
	FLEX.BAS	Flex interface file to the intelligence server.
Python	INT386W.PY	The python interface to the Intelligence server.
Delphi	INTELLIGENCESERVER.PAS	Function declarations for the INT386W.DLL
	FLEX.PAS	Flex interface file to the intelligence server.
.NET c# and VB.NET languages.	LPA.DLL	The .NET DLL interface to the INT386W.DLL
	FLEX.VB	Flex interface file to the intelligence server.
Java (The java files are in the directory: <prolog>\SERVER\INCLUDES\LPA\IS)	Int386w.java	Source definition for the Int386w class Java interface to the JINT386W.DLL
	JINT386W.DLL	The Java DLL interface to the INT386W.DLL
	FlexIS.java	Flex interface file to the intelligence server.