Querying Ergo from C and C++ Programs

Ergo has an API that allows C programs (and therefore C++ programs) to start the reasoner, load, add, and delete information within the running Ergo instances, send queries and obtain results returned by these queries. [A very extensive example](https://drive.google.com/a/coherentknowledge.com/file/d/1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq/view?usp=drive_web) that illustrates most of the important aspects is provided. This example has very detailed comments and here we just highlight the key points.

This example will also work in Flora-2, if flora\_query/5 is used everywhere instead of ergo\_query/5.

# Compilation

First, you need to write a C/C++ program, foo.c. It has to use the XSB/Ergo API described below, but how does one compile it and link with Ergo? Various options are described in the header to [the provided example](https://drive.google.com/a/coherentknowledge.com/file/d/1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq/view?usp=drive_web). Here is one of the simplest options:

**Linux**:

gcc -I*xsb-install-dir*/XSB/emu -I*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu \

-Wall foo.c -o foo \

*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu/saved.o/xsb.o \

-L*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu/lib \

-lm -ldl -Wl,-export-dynamic -lpthread

Make sure you replace the highlighted parts to match your system. The above links foo.c with XSB statically. For dynamic linking, use

gcc -I*xsb-install-dir*/XSB/emu -I*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu \

-Wall foo.c -o foo \

*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu/bin/libxsb.so \

-L*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu/lib \

-lm -ldl -Wl,-export-dynamic -lpthread

**Mac**: replace x86\_64-unknown-linux-gnu with i386-apple-darwin17.3.0 or whatever fits your version of MacOS X and use appropriate linker flags.

gcc -I*xsb-install-dir*/XSB/emu -I*xsb-install-dir*/XSB/config/i386-apple-darwin17.3.0 \

-Wall C\_calling\_Ergo.c -o C\_calling\_Ergo \

*xsb-install-dir*/XSB/config/i386-apple-darwin17.3.0/saved.o/xsb.o \

-L$HOME/XSB/XSB/config/i386-apple-darwin17.3.0/lib -lm -ldl -lpthread

For dynamic linking, change .so to .dylib.

**Windows**:

cl.exe foo.c /F10000000 /DWINDOWS\_IMP /I*xsb-install-dir*\XSB\emu \

/I*xsb-install-dir*\XSB\config\x64-pc-windows \

*xsb-install-dir*\XSB\config\x64-pc-windows\bin\xsb.lib

Make sure that the environment variable PATH is set so that the Visual Studio compiler, cl.exe, can be found through that variable.

In Linux, Windows, and the Mac, you can get the exact values to use for the directories like

*xsb-install-dir*/XSB/config/x86\_64-unknown-linux-gnu

*xsb-install-dir*/XSB/config/i386-apple-darwin17.3.0

*xsb-install-dir*\XSB\config\x64-pc-windows

By starting your copy of Ergo and issuing the query system{archdir=?D}.

**Running**:

foo *command-line-parameters*

# Key Points in Writing foo.c

The [provided example](https://drive.google.com/a/coherentknowledge.com/file/d/1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq/view?usp=drive_web) indicates the parts that must be kept as is or, if changed, must be kept equivalent, with the comments like KEEP THESE or DON’T CHANGE THESE. Each program that invokes Ergo must have these three parts:

1. Initialization of XSB
2. Initialization of Ergo
3. Sending commands and queries to Ergo, getting the results, exception handling

The included example parks these parts as PART 1, PART 2, and PART 3. You don’t need to understand the details of parts 1 and 2 -- simply copy them over to your program along with the variables and constants on which they depend. Other parts, marked as KEEP THESE and DON’T CHANGE THESE, should also be copied over. They include the basic C language setup and the “bridges” to XSB and Ergo.

**Sending queries and commands** to Ergo involves the use of the C function called xsb\_query\_string(), which returns XSB\_ERROR (if execution was unsuccessful), XSB\_SUCCESS (if the query had answers or was simply true), and XSB\_FAILURE (if the query had no answers and is false).

If a query has answers, they can be fetched, one by one, by executing the XSB function xsb\_next(), which also returns the same codes. The only difference is that it returns XSB\_SUCCESS if it managed to fetch the next answer and XSB\_FAILURE if no more answers can be fetched. When one finishes working with a query, it must be closed with xsb\_close\_query() before starting a new query.

**The structure of a query**. A query statement looks like xsb\_query\_string(*plgquery*); where *plgquery* has the form

**”**ergo\_query(**’***an\_ergo\_query***.’**, *variable\_map*,

*CompileStatusVar*, *UndefinedIndicator*, *ExceptionVar*),

*possibly\_other\_prolog\_subgoals***.”**;

Pay attention to the punctuation highlighted in red -- it is a must. Here ergo\_query() is a 5-argument predicate that sends the query *an\_ergo\_query* to Ergo and returns the results via *variable\_map* -- a structure that maps the Ergo variables in *an\_ergo\_query* to the XSB variables that can be accessed in the C program. The query *an\_ergo\_query* is a regular Ergo query and *variable\_map* is a list of the form [‘ergovar1’ = xsbvar1, ‘ergovar2’ = xsbvar2, …]. *CompileStatusVar* gets bound to a list that represents the status returned by the Ergo compiler after it is done compiling *an\_ergo\_query*. *UndefinedIndicator* is set for each query answer separately. If it is 0, the answer has the truth value True; otherwise, the truth value of the answer is Undefined (ie, the answer can be neither true nor false). False answers are not returned at all, of course. Finally, *ExceptionVar* gets bound to a descriptor of the exception that might have occurred during the runtime of the query. If no exceptions occurred, this variable is bound to the atom normal. Details about ergo\_query() can be found in [ErgoAI Reasoner User’s Manual](http://coherentknowledge.com/coherent-links/ergo-manual.pdf), Chapter on Passing Arbitrary Queries to Ergo.

The optional *possibly\_other\_prolog\_subgoals* can be any XSB goal that manipulates the variables of the main query. These are usually used to perform transformations that are easy to do in XSB but hard in C. Here is an example:

rc = xsb\_query\_string("ergo\_query('p(a(?X),b(?Y),c(?Z)).', ['?X'=X,'?Y'=Y,'?Z'=Z],

CompileStat, IsUndef, Exception), (\*)

flora\_get\_message\_from\_errorball(Exception, ExceptionMsg).");

Here *an\_ergo\_query* is p(a(?X),b(?Y),c(?Z)), and it has three Ergo variables ?X, ?Y, and ?Z. These variables are mapped by the *variable\_map* to XSB variables X, Y, and Z, respectively. In case we want to see the exception, we could have printed it directly, but if you tried you would notice that it contains all kinds of information. If we are interested only in the message part of the Exception, it can be obtained with the help of the XSB subgoal that is shown in (\*) right after ergo\_query(). See the [attached program](https://drive.google.com/a/coherentknowledge.com/file/d/1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq/view?usp=drive_web) for a complete example.

Edge cases concerning *an\_ergo\_query*:

* A backslash usually has to be quadrupled because of all the translations that happen by the time a command gets to the ergo compiler. For instance: writeln(abcde)@\\\\io.
* If a single quote must be inserted as part of *an\_ergo\_query*, it has to be doubled because *an\_ergo\_query* itself occurs inside single quotes.
* In Windows, don’t use backslashes inside path names -- use forward slashes.

**Setting parameters in a query**. If one needs to pass some values from the C program to Ergo, the easiest way to do so is during the construction of *an\_ergo\_query* string (e.g., using sprintf), as can be seen in the sample program. A more complex, but viable, way is to use

xsb\_set\_var\_int(Val,VarNumber);  
xsb\_set\_var\_float(Val,VarNumber);

xsb\_set\_var\_string(Val,VarNumber);

As the names suggest, these bind the variable number VarNumber (which must be a positive integer) to the value Val. For instance, xsb\_set\_var\_float(3+1.2, 2) would bind variable #2 to 4.2. In the above query (\*), we have seven variables: X (first), Y (second), Z (third), CompileStat (fourth), IsUndef (fifth), Exception (sixth), and ExceptionMsg (seventh). Therefore, Y will be bound to 4.2 and, by magic, that value will be passed to the Ergo variable ?Y into the query p(a(?X),b(?Y),c(?Z)) in (\*).

**Extracting variable bindings from a query answer**. After a query is executed, the variables in the query will be bound to the first answer. If these are primitive values int, float, or string, they can be fetched into the C program using the functions

(int)xsb\_var\_int(VarNum)

(char\*)xsb\_var\_string(VarNum)

(float)xsb\_var\_float(VarNum)

As before, VarNum is a positive integer that represents the desired variable. One must ensure that the variable number VarNum has the right type. In particular, using xsb\_var\_string() on a variable that is not bound to a string may crash the whole system.

Complex terms can also be returned to a C program, but this requires parsing such terms and we don’t recommend that. Instead, elementary values should be extracted on the Ergo/XSB side and passed back to C. Only a very advanced user should ever attempt parsing complex terms in C, which requires good familiarity with chapters "Embedding XSB in a Process" and "Foreign Language Interface" in [XSB Manual, vol. 1](http://coherentknowledge.com/coherent-links/XSB-manual-vol1.pdf).

The ergo interface does provide a small repertoire of functions that simplify certain common dealings with arbitrary terms:

xsb\_var\_to\_string(VarNum) // returns char\*

xsb\_var\_strequal(VarNum,String) // returns a Boolean: 0, 1

xsb\_var\_term(VarNum) // returns prolog\_term

The former fetches the binding of variable number VarNum and converts it to a C string. The second takes the binding of variable number VarNum, converts it to a string, and then compares it to String. If they are equal, 1 is returned; 0 is returned otherwise. The third function returns the binding of variable number VarNum as a general term. This is provided for advanced users who dare to do term parsing in C.

Once the current answer has been processed in C, the user can execute xsb\_next() to fetch the next answer. If xsb\_next() returns XSB\_SUCCESS, it means the next answer has been obtained. Processing that next answer is the same as just described.

See the [provided example](https://drive.google.com/a/coherentknowledge.com/file/d/1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq/view?usp=drive_web) for the use of all these functions in a context.

**Error handling**. When calling xsb\_query\_string() and xsb\_next() (and, preferably xsb\_close\_query() as well), the return code must be checked. If they return XSB\_ERROR, the operation was unsuccessful and trying to fetch or process query answers is likely to crash the program due to memory violation.

However, even if these commands return XSB\_SUCCESS or XSB\_FAILURE, it does not mean that the Ergo query in question succeeded. As we have seen in (\*), ergo\_query() has the Exception argument, which we can check. For our example query (\*) above, one should do:

if (!xsb\_var\_strequal(6,"normal")) // var #6 is Exception  
 fprintf(stderr, "query has exception: %s\n", xsb\_var\_to\_string(7)); // var #7 is ExceptionMsg

else if (rc == XSB\_SUCCESS) // recall: rc gets the return code of the query in (\*)  
 do {  
 // process query results

… … …   
 } while (xsb\_next() == XSB\_SUCCESS); // get next answer

else  
 printf("query has no results\n");

Here !xsb\_var\_strequal(6,"normal") checks if there was an exception and if so prints an error message. Otherwise, if there are answers they are processed; if the query has no answers, a message to that extent is printed.

Finally, the CompileStat argument to ergo\_query() contains a list that indicates whether the Ergo query was compiled without errors. It usually has the form [eof,success] or [eof,failure]. This is rarely used because failure to compile causes an exception that is also reported via the Exception variable.

Exception handling is illustrated in the [provided example](https://drive.google.com/a/coherentknowledge.com/file/d/1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq/view?usp=drive_web).