An Improved C Calling Interface for Unicon Language

Udaykumar Batchu

Abstract

This report describes an improved interface for calling library or user written C functions from Unicon. It eliminates the need for writing wrapper code, whenever external C functions are used in Unicon programs. This paper explains the design and implementation of the new interface.

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003

Advisor: Dr. Clinton Jeffery

1. Introduction

Unicon is a high-level programming language which inherits most of its features from Icon, a project initiated at the University of Arizona. Unicon is elegant, portable, expressive and platform-independent which makes it a perfect choice for developing various applications such as objects, networks and databases [Jeffery03].

Some of the features of the Unicon language include easy syntax, no type declarations for the variables to be used, powerful string manipulation functions not available in other languages such as C, C++ or Java, high level graphics abilities along with object-oriented facilities [Jeffery03] and robust support for various input-output functions.

Many programmers need to use their existing C code, in languages such as Unicon. This was supported by a function called `loadfunc()` in Icon and Unicon which loads external C functions dynamically at runtime. However, `loadfunc()` is little used, largely because it requires programmers to write wrapper code by hand, to convert parameters and return types between C and Unicon and create the shared library containing the C function and its wrapper for use with `loadfunc()`.

This project simplifies the process with a new interface to the Unicon language for loading external C functions easily with only a little effort from the programmer. For this two new preprocessor directives `$c` and `$cend` are introduced. Using these new directives, C function signatures can be declared inside a Unicon program along with the name of the library from which they are being used. The whole process of generating the wrapper code to loading the shared library at runtime using `loadfunc()` is automated.

The following section gives an introduction to the Unicon language itself and people who are already familiar can skip to the next section. Section 2 talks about the current mechanism for loading external C functions and the steps necessary for doing that. Section 3 explains the design and implementation issues involved in developing the new interface. Section 4 gives insight into the interface macros used for writing the wrapper

code. Section 5 gives details of how the shared library is built which is used for loading the external C functions. Generation of Icon/Unicon stubs and passing them back to Unicon is given in Section 6. After that, results of this new interface are explained in Section 7. Section 8 looks into the similar work done in other programming languages. Finally, Section 9 gives a conclusion and brings up the directions for future work.

1.1 Unicon Language Overview

Basic Data Types

Unicon supports various primitive data types such as integer, real, strings and a special type called Csets. The following details of Unicon types are taken from [Jeffery 03].

- **Integers-** Unicon represents integers using the C long type and they are of arbitrary precision.
- **Real -** "Reals are double-precision floating-point values".
- **Strings -** String literals are declared pretty much the same as in C, but Unicon's string type is immutable.
- **Csets -** Csets are sets of 0 or more characters specifically called as character sets. They are used in string analysis and pattern matching. Csets are represented within single quotes like '`aeiou`'. Csets can include escape sequences for encoding special characters.

Data Structures

Unicon provides a variety of structures for storing, manipulating and accessing different kinds of data. Depending on the problem at hand, the data structures supported within Unicon like lists, tables, records and sets can be used. Importantly, all Unicon structures can store heterogeneous values.

- **Lists -** The Unicon list data type provides data structures programmers often have to implement themselves in C. Such as dynamic-sized arrays, linked lists, queues and stacks. Built-in functions operate on lists for addition, deletion and insertion of elements.

Tables - Tables are a form of associative arrays where elements are accessed by their key instead of an array index. There is no corresponding type in C, the nearest common data structure is a hash table which is not standardized in C.

- **Records -** Records are a collection of named fields, similar to structures in C language. A record is declared as follows:

  ```
  record complex(re, im)
  ```

  Fields do not have declared types, although type conventions are usually followed.

- **Set -** A set is a collection of distinct elements considered as a whole [Wiki]. There is no restriction on the type of elements that can be present in a set. C has no matching data type.

Procedures

A procedure starts with the keyword **`procedure`** followed by the name and then the parameters it takes within the braces "( )". A sample procedure, calculating the sum of n numbers can be written as follows:

```
procedure sumofNnumbers(n)
     return (n * (n+1))/2
end
```

Parameters and return values do not have their types defined. The parameters passed to a procedure are pass by value except for structures are passed by reference.

1.2. Current Mechanism - Dynamically Loading C Functions: loadfunc

In order to call external C functions in Unicon program code, a program must first load that code by calling built-in function called *loadfunc ( )* which takes two arguments. The definition looks as follows:

loadfunc (filename, funcname)

where **filename** is the object library name from which the external C function is loaded. Both the arguments are strings and the second argument **funcname** is often the name of a wrapper function which handles the data type conversions from Unicon data types to C data types and vice versa[IB02]. Importantly loadfunc uses the concept of dynamic loading and linking; the related functions are generally found in the 'dlfcn.h' header file under C language. The actual code for loadfunc is placed in /unicon/src/runtime/fload.r file.

Example for Loading a C Function using loadfunc

The programmer can explicitly call `loadfunc` from within a Unicon program with a proper library filename to look for the external C function inside it [IA36].

e.g. An example program: **bcount.icn**

```
procedure main()

   local i
   bitcount := loadfunc("/unicon/bin/libcfunc.so", "bitcount")

   every i := 250 to 260 do
      write(i, " ", bitcount(i))

end
```

In the above program, "`libcfunc.so`" is the library file, that contains the C function `bitcount` (calculates the bit count of an integer) to be loaded dynamically at runtime using `loadfunc`.

Converting between Unicon and C data types

When a programmer wants to use a particular C library function or their own C function, they have to write the wrapper code consisting of the data type conversions for all the

parameters involved in calling that C function. Currently Unicon only supports data type conversion for basic data types such as: integers, real numbers, files and strings. These conversion macros are found in /unicon/ipl/cfuncs/icall.h.

The /unicon/ipl/cfuncs directory contains some sample C functions to use in Unicon programs, without the user writing any extra piece of code (wrapper code). For calling a new external C function from Unicon programs, there are a variety of steps that need to be followed. We shall demonstrate the mechanism in the next few lines.

Suppose the user (programmer) wanted to use the following fibonacci function in Unicon. The steps to be followed are:

Step 1) Write or locate the C code to be called. For example, here is C code for Recursive Fibonacci Series:

```
int fib(int n)
{
        if(n<2) return 1;
        else
                return fib(n-1) + fib(n-2);
}
```

Step 2) For the C function(s) to be called, the programmer needs to write wrapper code in C for data type conversions using the "icall.h" header file. For example, the following function is wrapper code for Recursive Fibonacci series.

```
int fibwrap(int argc, descriptor *argv) /*: Fibonacci */
{
        unsigned long v;
        int n;

        ArgInteger(1);

        v = IntegerVal(argv[1]);
        n = fib(v);
        RetInteger(n);
}
```

Step 3) Build a library - The wrapper code along with the original C code can be written in the same file, say: 'fibonacci.c'. In the present mechanism, programmer creates the object file (fibonacci.o) and supplies it to a Makefile which in turn adds it to the 'libcfunc.so' library.

Step 4) Write an Icon/Unicon stub procedure to load and call the wrapper function. Insert this procedure in a related Unicon (*.icn) file. This procedure uses '`pathload`' which calls `loadfunc` to load the particular C function dynamically at runtime. The code looks as follows:

```
# fibonacci.c:
procedure fib(a[])     #:Fibonacci
   return(fib:=pathload(LIB,"fibwrap"))!a;end
```

Step 5) Link in the Stub Procedure - After all the above steps have been performed, if a programmer wants to use 'fibonacci' inside a Unicon program, then a link to the file containing the stub is required. An example program looks as follows:

e.g. fibnum.icn

```
link cfunc         # contains stub procedures

procedure main()

       i :=  fib(6)
       write(i)
end
```

2. Improved Mechanism - Design

Since the programmer is burdened with writing the wrapper code along with several other steps whenever a particular external C function is used, the new interface makes it easier for the Unicon end user to call C functions with ease.

The new mechanism introduces two new preprocessor directives '`$c`' and '`$cend`' for declaring the external C functions that are going to be used in a Unicon program. When a

Unicon program is compiled, the preprocessor scans the source file for any defined preprocessor directives for replacement of the code and when '`$c`' and '`$cend`' are encountered, the whole C code present inside those two constructs is grabbed and taken for further processing [IU04].

The new mechanism defines a standard format in which the Unicon programmer should declare their external C functions which are going to be used in the program. Later in this document, several examples are provided for declaring external C functions using the new directives.

Function signatures should be provided for all the external functions being used inside a Unicon program. The signature contains the name of the library (object file) from which the C function is loaded, followed by a function prototype with all its parameters, and return types as used in standard C language. More functions can be declared by separating them with a comma and the same format is followed for declaring more libraries and the functions being used from them.

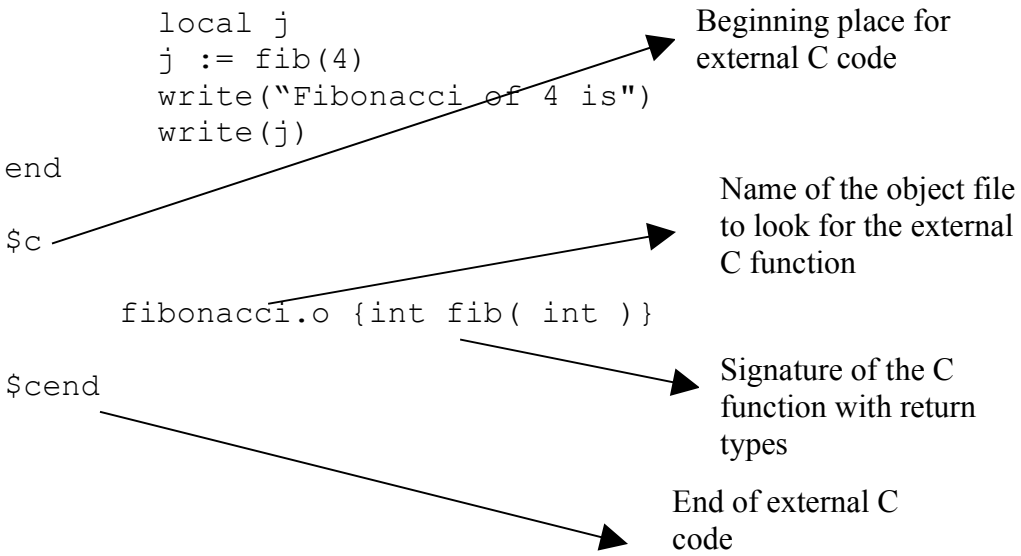
The preprocessor grabs the contents within the constructs `$c` and `$cend` as a big string and passes it to a function called *cincludesparser*, and parses it to automatically generate the corresponding wrapper code for each of the external C function used. It then builds a procedure which uses loadfunc to load the external C functions dynamically at run time.

2.1 An Example Format: unifib.icn

```
procedure main()
        local j
        j := fib(4)
        write("Fibonacci of 4 is")
        write(j)
end

$c
      fibonacci.o {int fib( int )}

$cend
```



In the above example program, 'fib' is being called inside the Unicon code. It is declared to be an external C function found in "fibonacci". Any number of functions can be used from a single C source (object) file and those multiple function signatures should be separated by commas enclosed in curly braces. Also within the '`$c`' and '`$cend`' constructs one can declare as many external object files with function signatures to look for external C functions.

2.2 Execution of the Design - Wrapper Code Generation – *`cincludesparser`*

Once the whole set of external function signatures are passed to the *cincludesparser* function as a string, scanning is done on it to extract the names of the libraries and the details of C functions being used. Unicon lists of various dimensions and sizes are used to hold the library names, C function names, return types of the functions and return types of the

actual function parameters. The following are the key lists involved in wrapper code generation:

- *libname* - a list containing the names of the libraries being used inside Unicon
- *funret_type* - two dimensional list holding the return type of C functions
- *funname* - two dimensional list having the names of C functions being called
- *arg* – three dimensional list comprising the return types of the function parameters and the parameter count

After all the parsing is done on the string passed to the function, full information regarding external C functions is stored in the respective lists. As a next step, this same function writes out wrapper code(s) which are also called loadable C functions [IA36]. The prototype of a loadable C function looks as follows:

```
int funcname(int argc, descriptor *argv)
```

`argc` - is the number of arguments
`argv` - is an array of descriptors, which are structures holding the Unicon values passed
`funcname` - is the name of the wrapper code being written for the actual C function

The above mentioned function prototype is very similar to passing command line arguments to a C `main()` function. **`argv[1]`** to **`argv[argc]`** are the actual arguments that are passed from Unicon function call to the original C function. **`argv[0]`** is used to store the return value on success or the offending value in case of error. When writing out the wrapper code, the name of the wrapper function *funcname* is derived from the original C function with a *wrap* appended at the end of it for easier tracking. Also the filename of the wrapper code is related to its original C file (library) with a *wrap* appended to its end making it another new C program file having "**.c**" as its extension. Wrapper code for all the external C functions belonging to a particular library are written to a single file for clear manageability.

After all the statements present in the *cincludesparser* function are executed, the automatic generation of wrapper code for all the external C functions will be done with creation of new C files in the current directory. These new wrapper C files are helpful in building the shared library in the next stage along with creation of Unicon/Icon stubs needed for C function calling.

3. Interface Macros

There is a C header file present under the `/unicon/ipl/cfuncs` directory named "`icall.h`" containing the macros needed for converting Unicon values to C values and viceversa. A `-I` option to the C compiler helps in finding the header file in the specified path. As part of this project, support for handling C arrays (single dimensional) is provided and the corresponding macros can be found in the same header file mentioned above. These macros are a result of the generous work done by Mr. Kostas Oikonomou, AT&T. A whole set of different macros present in the "`icall.h`" header file provides support for various data types such as integers, reals, characters, single dimensional arrays and little support for file values [IA36]. Responsibilities of the macros include:

- knowing about the data type of Unicon value passed
- validating the type of an argument, Unicon value instead
- converting the Unicon value into an equivalent C value
- returning back the C value back to Unicon after C function calling is done
- error handling during data type conversion

All the macros listed below are used in writing out wrapper code functions for all the external C functions that are going to be called from Unicon. Depending on the data type of C function arguments and return types, matching macros are called to get the job done. The complete list of macros available for help in writing wrapper code is mentioned in Appendix A.

3.1 Wrapper Code Generation Using Macros - Simple Data Types

This section presents a small example Unicon program using external C functions which are being loaded with the help of the new preprocessor directives `$c` and `$cend`. Then we examine the generated wrapper code for a particular C function.

e.g. mathchar.icn

```
procedure main()

      local a, str  # variable declaration

      write("enter a value for its factorial: ")
      a := read()      # reading the input
      write("enter a string to check for palindrome: ")
      str := read()    # asking the input for a string

# the following statements are the calls to C functions
write("the factorial of ", a, "is: ", factorialfunc(a))
write("*******the check for palindrome******* ")
palindrome(str)
end

$c
      fact.o { int factorialfunc( int) },
        checkpalin.o { int palindrome( char) }
$cend
```

This program is using external C functions present in the libraries '`fact.o`' and '`checkpalin.o`' each one having a function in it. When we clearly look into the format in which the two functions are declared, each library is on a new line with functions belonging to a particular library begin with its name at the start followed by the function signatures inside curly braces at the end. When the above Unicon program is run at the command line, as a first step the preprocessor scans the source file for any preprocessor directives to replace the code and also performs any other actions given.

During this compilation stage, once the `$c` and `$cend` directives are encountered, wrapper code is automatically generated for the mentioned external C functions. In the

example program above mentioned, the wrapper code for `factorialfunc` looks as follows:

`factwrap.c` - the wrapper code for `factorialfunc`

```
#include <stdio.h>
#include "icall.h"
int factorialfunc(int);

int factorialfuncwrap(int argc, descriptor *argv)
{
        int returnValue;
        long arg1;

        ArgInteger( 1 );
        arg1 = IntegerVal( argv[ 1 ] );
        returnValue = factorialfunc( arg1 );
        RetInteger( returnValue );
}
```

3.2 Wrapper Code Generation Using Macros - Single Dimensional Arrays

Support is provided for both integer arrays and real (double) arrays parameters. This feature is illustrated by a sample Unicon program using a C function which takes an array as a parameter. The generated wrapper code for functions involving arrays is presented. The following example code explains a few details of how an external C function involving arrays is called:

e.g. testarrays.icn

```
procedure main()
      local elements, loop, size
      local dblelements, avg

      write("enter the size of the array: ")
      size := integer(read())
      elements := list(size, 0)
      dblelements := list(size, 0)

      write("enter the elements for sorting")
      every loop := 1 to size do {
            elements[loop] := integer(read())
```

```
      }
      write("enter the reals(doubles) elements to find average")
      every loop := 1 to size do {
            dblelements[loop] := real(read())
      }
      avg := Caverage(dblelements, size)
      bubbleSort(elements, size)

      write("***********External C Functions************\n")
      write("the list after getting sorted with bubbleSort is:")
      every write(!elements)
      write("the average of the list of doubles is: ")
      write(avg)
end
$c
            bubble.o { void bubbleSort(int[],    int ) },
             arrayavg.o { double Caverage(double[], int) }
$cend
```

The above program, calls two external C functions `bubbleSort` and `Caverage` from two different `.o` object files, namely `bubble.o` and `arrayavg.o`. These two functions are declared with the help of the new preprocessor directives `$c` and `$cend`. The format for declaring a function parameter to be a single dimensional array is to suffix the data type with array indices '`[]`'.

When the `cincludesparser` scans the C functions, the array indices '`[]`' at the end of a type declaration declares the parameter type to be an array and causes appropriate wrapper code to be generated. Here, the automatically generated wrapper code for `bubbleSort` and `Caverage` used in the sample program looks as follows:

`bubblewrap.c` - the wrapper code for `bubbleSort`

```
#include <stdio.h>
#include "icall.h"
```

```
void bubbleSort(int[], int);

int bubbleSortwrap(int argc, descriptor *argv)
{
        /* the return type is void */

        int *arg1;
        long arg2;
        word r; /*Array declaration done*/
        ArgList( 1 );
        arg1 = (int *) malloc((int) ListLen( argv[ 1 ] )* sizeof(int));
        IListVal( argv[ 1 ],arg1 );
        ArgInteger( 2 );
        arg2 = IntegerVal( argv[ 2 ] );
        bubbleSort( arg1, arg2 );

        IValList(a1, argv[1]);
}
```

`arrayavgwrap.c` - the wrapper code for `Caverage`

```
#include <stdio.h>
#include "icall.h"
#include "arrayavg.c"

int Caveragewrap(int argc, descriptor *argv)
{
        double returnValue;
        double *arg1;
        long arg2;
        word r; /*Array declaration done*/
        ArgList( 1 );
        arg1 = (double *) malloc((int) ListLen( argv[ 1 ] )*
sizeof(double));
        RListVal( argv[ 1 ],arg1 );
        ArgInteger( 2 );
        arg2 = IntegerVal( argv[ 2 ] );
```

```
        returnValue = Caverage( arg1, arg2 );
        RetReal( returnValue );
}
```

The above wrapper functions clearly show the difference between the macros used for integer arrays and real arrays. Depending upon the function returning a value or just returning nothing like the `void` type, the call to the original C function varies inside the wrapper code.

```
avg := Caverage(dblelements, size)
bubbleSort(elements, size)
```

Since the `Caverage` function returns a real variable as it is known from the function signature provided, there is a variable `avg` declared to store the returned value at the Unicon level. But when we look at the other C function call to `bubbleSort`, the function returns nothing since it is declared `void`. But, after the call to `bubbleSort` is done, the contents of the array passed to the C function are changed. Call to `IValList` macro helps in copying back the modified array contents into the original Unicon list that has been passed to C wrapper function.

Here when delved at the calls made inside the wrapper code, the first step as usual is validating the type of the argument passed, an array type with `ArgList` and then create a corresponding C array of integers or reals (doubles) by allocating memory using the built-in `malloc()` function. As a next step, copy the contents of Unicon list elements passed from the Unicon program into the newly created C array using `IListVal` or `RListval` correspondingly, the first being for integers and the other being for reals. These newly created arrays are then used as parameters to call the original C function.

The interface macros for handling single dimensional C arrays do not actually reflect the changed contents inside the Unicon lists which were passed for calling the C functions. For preserving the pass by reference nature of C arrays as well as Unicon lists, wrapper

function needs to copy back the modified array back into the Unicon list, by using the macro calls `IValList` or `RListVal` for handling integer and real lists accordingly.

4. Building the Shared Library

Whenever external C function(s) are to be used in Unicon, all these functions must be present in the form of a shared object file to load them dynamically at run time [IA36]. The process of building the shared library starts with having the original C source files or object files from which the function is being used along with the automatically generated C wrapper code file(s).

4.1 Earlier Method

The earlier mechanism involved a lot of manual intervention in creation of the shared library `libcfunc.so` found under `/unicon/ipl/cfuncs` directory. This shared object file was then used by `pathload()` to load the external C functions dynamically at runtime [MAN]. As part of this project, the whole process is being automated which was earlier dependent on the external Makefile changes and shell scripts.

4.2 Improved Mechanism - *make_library*

In the new improved mechanism, the responsibility of building the required shared library became a part of the preprocessor and the dependence on UNIX shell script is eliminated. A new function called `make_library` is introduced to do the job of building the shared object file. The working of this function is similar to the working of aforementioned shell script. Only change is that, this function is pure Unicon code and can be easily ported to other platforms such as Windows and the knowledge of shell scripting is no longer needed.

This new function `make_library` takes a list of libraries as an argument helpful in building the shared library. Once the generation of wrapper code files for all the external C functions is completed by the `cincludesparser` function, it creates a list

containing both C library files (source files) and the newly created compiled wrapper files and passes it on for building the shared library. The C wrapper files created are compiled first [IA36] using the following command

```
cc -c factwrap.c
```

which in turn produces an object file `factwrap.o` instead of an executable program. Same procedure is followed for all the wrapper code files created and are then used for building the shared library (shared object) file.

While building the shared object file, the naming convention followed depends on the name of the Unicon source file. If the name of Unicon program is, for example: '`test.icn`', name of the shared object file will have a character 'c' prefixed with the extension being changed to **`.so`** during its creation. Finally becomes *`ctest.so`* and is used for dynamically loading the C functions with the help of `loadfunc`.

5. Generation of Icon/Unicon Stub

When a function call is made to an external C function inside Unicon, the compiler will search for the corresponding function definition assuming that it is declared as a procedure within the program. So in order to make the external C function call look like a call made to a Unicon procedure defined within the program or a built in Unicon function call, we bind the actual C function as a Icon/Unicon procedure.

When the preprocessor finds `$c` and `$cend` directives where information about the external C functions are given, it generates the corresponding *Icon/Unicon* stubs which have the same name as the original C function for dynamically loading them at run time. When the required wrapper code and the corresponding shared library are built for loading external C functions, the `cincludesparser` function presented earlier generates the necessary *Icon/Unicon* stubs (procedures) as a big string and passes them back to the Unicon source code and appends it to the end of the program. Since `pathload()` (which in turn uses `loadfunc()`), is used for loading a program from a library (shared) which is present under `/unicon/ipl/procs/io.icn` file

containing procedures for input and output, hence the statement '`link io`' is also passed back along with the Icon/Unicon stubs.

```
$c
            bubble.o { void bubbleSort(int[],      int ) },
            arrayavg.o { double Caverage(double[], int) }
$cend
```

In the above two external C function declarations mentioned in the aforementioned example program `testarrays.icn`, the Icon/Unicon stubs generated and passed back to the Unicon program look as:

```
link io
#bubbleSort.c
procedure bubbleSort(a[])                #calculates bubbleSort
return(bubbleSort:=pathload("ctestarrays.so","bubbleSortwrap"))!a;end
#Caverage.c
procedure Caverage(a[])          #calculates Caverage
return(Caverage:=pathload("ctestarrays.so","Caveragewrap"))!a;end
```

These two Unicon procedures reveal that a call to `bubbleSort` or either `Caverage` is like just another simple call to a procedure defined within the Unicon source program, hence hiding the details of external C function calling. The procedure call to **`pathload(libname, funcname)`** [IA36] which in turn calls `loadfunc()` then loads the function `funcname` from the library `libname` which it searches for in the paths mentioned by the **FPATH** environment variable.

6. Results

The improved interface for calling external C functions is successfully implemented for handling basic data types such as integers, double and characters along with the integer and double single dimensional arrays. The project was developed under the 32-bit Linux

(Kernel Version 2.6.11.4) environment, available within the Computer Science Department of NMSU.

All the work done related to the interface and the corresponding code written might work under any Linux or Solaris or Windows platform with the Unicon language being set up depending on the underlying platform. As the platform changes, building the shared library for loading the external C functions also changes. When compared to the earlier mechanism of using external C functions inside Unicon to the new interface, the significant changes are:

- Implemented a C language interface for Unicon for easy calling of C procedures/functions within Unicon programs.
- Introduction of new preprocessor directives `$c` and `$cend` to handle the external C code included.
- Automatic generation of wrapper code for handling the C data types and converting them from C data types to Unicon and vice versa. Modifying the Unicon language to deal with the external C code during compilation stage.
- A considerable reduction in Unicon programmer's effort in using already existing C procedures or functions within their Unicon code.

Calling Built-in C Functions*

Calling built-in C functions such as the ones provided in libraries `<math.h>` , `<string.h>`, `<ctype.h>` or `<stdlib.h>` is easier with very few lines of C code being written into a file and calling them inside Unicon. A few examples of calling built-in C functions `atoi` and `pow` from Unicon looks as follows:

e.g. External C function declarations in `test_clib.icn` program

```
$c
$include "/usr/include/math.h"
```

```
$include “/usr/include/stdlib.h”
            /usr/lib/libc.so { int atoi (char *) },
              /usr/lib/libm.so { double pow ( double, double) }
$cend
```

The two $include statements within the new preprocessor directives help in locating the definitions of external C built-in functions. Wrapper code for loading these C functions is the same as seen in the above mentioned examples and look as follows:

e.g. Wrapper code for `atoi`

```
#include <stdio.h>
#include <stdlib.h>

int atoiwrap(const char *n)
{
      return atoi(n);
}
```

e.g. Wrapper code for `pow`

```
#include <stdio.h>
#include <math.h>

double powwrap(double a, double b)
{
      return pow(a, b);
}
```

The C interface makes it a lot easier to call built-in C functions in the above mentioned manner. The time taken to generate the wrapper code, build the shared library and return the final Icon/Unicon stubs back to the Unicon program for dynamic loading depends on the number of external C functions pre-declared. The fewer the number of C libraries (functions) declared inside a Unicon program, fast is the process of all the aforementioned steps and ready for loading C functions dynamically at runtime.

Whenever header files information is seen, the functions are considered to be built-in C functions and the Icon/Unicon stubs generated for them have the same C function name to make it easier for using them. Stubs generated for the above two functions look as:

```
procedure atoi(a[])              #calculates atoi
        return(atoi:=pathload("ctest_clib.so","atoiwrap"))!a;end


procedure pow(a[])              #calculates pow
        return(atoi:=pathload("ctest_clib.so","powwrap"))!a;end
```

Calling built-in C functions with a little effort along with using user written C functions has been made a lot easier with elimination of a number steps when compared to the previous mechanism. Writing the wrapper codes themselves, compiling them and updating the corresponding Makefile and run it to create a new or updated shared library: all these steps are wiped out and a great deal of programmer's time is saved with this new interface. Even while using external C functions with the help of new preprocessor directives, there is only a few milli seconds of difference in the whole execution of the Unicon program.

7. Related Work

The concept of calling other language code inside a language is very old. Programmers always prefer to use their existing code inside a new language environment for reusability, compatibility and saving a lot of time. This section describes some of these interfaces available for use now.

SWIG

The SWIG (Simplified Wrapper Interface Generator) tool was originally developed at the Los Alamos National Labs and is currently being supported by the University of Chicago. SWIG currently supports a wide variety of languages for wrapper code generation ranging from Perl, PHP, and Python to Java, C#.

The SWIG manual says that "*SWIG is an interface compiler that connects programs written in C and C++ with scripting languages such as Perl, Python, Ruby and Tcl"* [SWIG]. There a lot of tasks such as string scanning, regular expressions and pattern matching which can be done a lot easier in scripting languages than in regular

programming languages such as Java or C#. Also there are many built-in functions provided with the native languages such as C/C++ which cannot be found in scripting languages and this is the place where SWIG acts like a bridge between these two.

We now examine a small Perl program which is trying to use functions written in C language in it. In order to do this, we need to have the source C program, the interface file *which is the input to SWIG* [SWIG]. Finally we compile the C source file and build a shared library for use inside our Perl program.

Step 1) C source file.

e.g. example.c

```
#include <time.h>
double My_variable = 3.0;

int fib(int n) {
     if (n <= 1) return 1;
     else
            return (fib(n-1) + fib(n-2));
}
int my_mod(int x, int y) {
     return (x%y);
}
char *get_time()
{
     time_t ltime;
     time(&ltime);
     return ctime(&ltime);
}
```

Step 2) Interface file

Whenever we want to use this C language code in foreign languages, an interface should be created and this file helps as one. This interface file is nothing but the Icon/Unicon stub generated by the *`cincludesparser`* function, which uses `pathload` function to dynamically load the C functions at run time.

e.g. example.i

```
%module example
%{
 /* Put header files here or function declarations like below */
%}
extern double My_variable;
extern int fib(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

Step 3) Building a Perl module

The following are the sequence of commands issued on Solaris machine by using Perl5. SWIG creates the wrapper file `example_wrap.c` and `gcc` and `ld` builds the shared library `example.so` for use with Perl program.

```
unix % swig -perl5 example.i
unix % gcc -c example.c example_wrap.c \
        -I/usr/lib/perl/solaris/5.003/CORE
unix % ld -G example.o example_wrap.o -o example.so
unix % perl
use example;
print $example::My_variable,"\n";
print example::fib(8),"\n";
print example::get_time(),"\n";
<ctrl-d>
3.0
21
Tue Nov  1 13:42:38 MST 2005
unix %
```

Due to space and time constraints, more details about using SWIG is not possible. For learning and usage of SWIG which is available for free download from its website www.swig.org is a good place to start.

JNI - Java Native Interface

The Java Native Interface JNI is provided as part of the Java programming language toolkit.

Users of the Java programming language can use *native code* written in languages such as C/C++ inside their Java programs with ease by the help of JNI. The following figure demonstrates the role of JNI [Liang]:

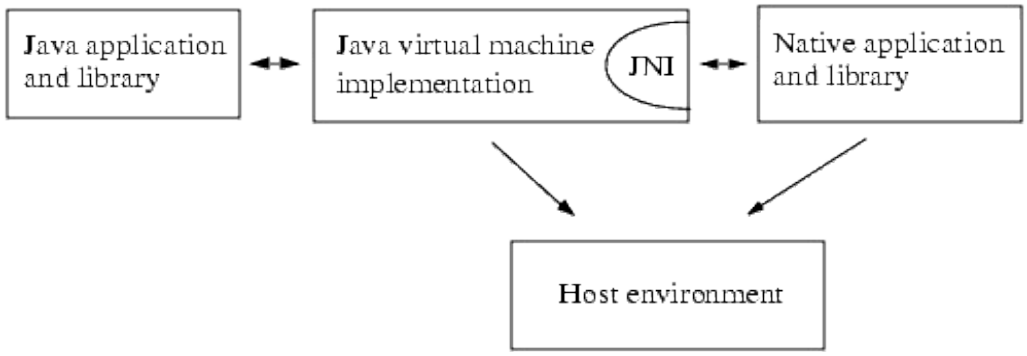


Figure 1.1 Role of JNI

JNI acts as a bidirectional interface allowing Java applications to call code written in other languages and vice versa with the help of its virtual machine. There is a visual interpretation of using JNI [Gloster] in the following figure.

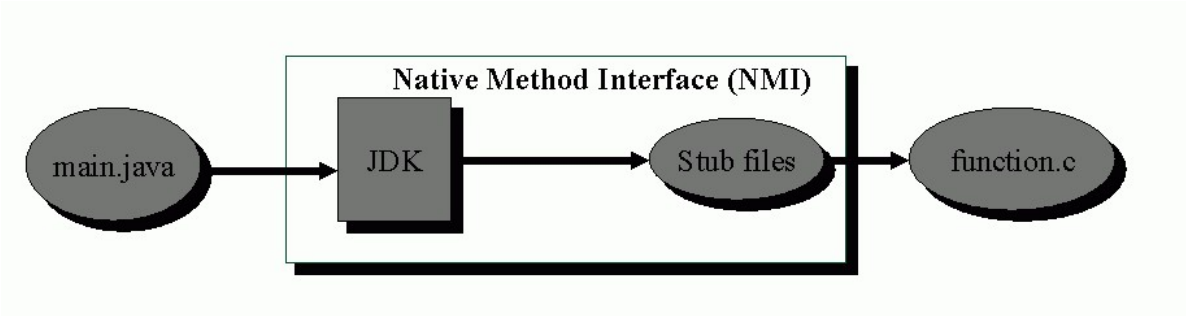


Figure 1.2 Calling C/C++ functions from Java using JNI.

There are a few steps to be followed before start using C/C++ code inside Java code.

Step 1) Create the Java source file which is going to use the *native method* [Liang].

e.g. Fibonacci.java

```
class Fibonacci {
     private native int fib();
```

```
     public static void main(String[] args) {
        int y;
            // Create an object of class Fibonacci
            Fibonacci findfib = new Fibonacci();
            /* convert the command line argument to a integer */
            x = new Integer(args[0]).intValue();
            y = findfib.fib(8);
            System.out.println("Fibonacci value of" + x + "is : " + y);
            }
    }
     static {
         System.loadLibrary("Fibonacci");
    }
}
```

The `Fibonacci` class mentioned above uses the a method `fib()` from the C language which is then followed by an instantiation of `Fibonacci` class which in turn is calling the `fib()` method.

Step 2) Compiling the Java source file to create the `Fibonacci.class` file

```
javac Fibonacci.java
```

Step 3) Generating the interface header "`Fibonacci.h`"

```
javah -jni Fibonacci
```

The header file created contains the function prototypes necessary for native method invocation [Liang]. The header file created in this step and the following native method implementation act as the interface between Java code and C programs.

Step 4) Native Method Implementation

```
#include <stdio.h>
#include <jni.h>
#include "Fibonacci.h"
#include "fib.c"
JNIEXPORT jint JNICALL
Java_Fibonacci_fib(JNIEnv *env, jobject obj, jint x)
{
      if( x < 2) return 1;
      else
            return (fib(n-1) + fib(n-2));
```

```
}
```

The C program includes three header files [Liang]:

- `jni.h` -- This header file provides information the native code needs to call JNI functions. When writing native methods, we must always include this file in our C or C++ source files.
- `stdio.h` -- The code snippet above also includes `stdio.h` because it is the standard header file to be included for all C programs.
- `Fibonacci.h` -- The header file that we generated using `javah`. It includes the C/C++ prototype for the `Java_Fibonacci_fib` function.

Step 5) C Source Compilation and Building the Native Library

Depending on the underlying platform on which we are running Java/JNI, the commands needed for building the shared library change. The sample commands needed for building a shared library on Solaris [Liang] are:

```
cc -G -I/java/include -I/java/include/solaris \
     Fibonacci.c -o libFibonacci.so
```

Because of the constraints posed by this page width we are actually seeing the command in two consecutive lines instead of one. The above command finally builds the shared library named **`libFibonacci.so`** ready to use with the Java program.

Step 6) Executing the Java Program

```
java Fibonacci
```

when the above command is run at the command line, it displays the value `21`, the 8th Fibonacci series number on the output window.

More notes and tutorials on how to use JNI can be found on the sun website (http://java.sun.com/docs/books/jni/). JNI is another good tool for developers to learn more about interfacing external programs/procedures into their own native languages.

Python and C

Even Python language supports calling external C functions within its code with the help of an interface. To call C functions from Python, the user has to write a Python extension

module in C. This extension module acts as an interface between the Python program and C function. For doing this Python provides an API "`Python.h`" for writing extension modules. The extension module must include the code of C function. The user has to import this extension module inside the Python program and call the C function as a method.

For example, if the user wants to call a C function `fib(n)`, which finds the n^{th} Fibonacci number, the user has to import the extension module `Fibonacci` containing `fib` and then call it.

```
import Fibonacci
Sum_fib = Fibonacci.fib(n)
```

The implementation of the extension module `Fibonacci.c` is shown below:

```
#include <Python.h>

int find_fib(int n)
  {
      if (n < 2) return 1;
      else return (find_fib(n-1) + find_fib(n-2));
  }

static PyObject* fib_sum(PyObject *self, PyObject *args)
{
    int a;
    int s;
    if (!PyArg_ParseTuple(args,"i",&a))
       return NULL;
    s = find_fib(a);
   return Py_BuildValue("i",s);
}

static PyMethodDef ModMethods[ ] = {
    {"fib", fib_sum, METH_VARARGS, "Description.."},
    {NULL,NULL,0,NULL}
};

PyMODINIT_FUNC initFibonacci(void)
{
    PyObject *m;
    m = Py_InitModule("Fibonacci",ModMethods);
    if (m == NULL)
       return;
}
```

The first line `#include <Python.h>` pulls the Python API. There are 3 steps in creating an extension module. The first step involves implementation of all the methods that will be called from the Python source program. The second step involves declaration of the list of all methods. The third step involves initializing the extension module.

Implementation of the Methods

For each of the C function being used inside Python, there should be a method declared in the Python extension module [UA]. Here, the function fib_sum( ) in Fibonacci.c calls the function find_fib( ) which has the implementation for finding n^{th} Fibonacci number. In the first step, fib_sum( ) converts the arguments passed from Python program into C data types using the function PyArg_ParseTuple( ). Then the function find_fib( ) is called which returns nth Fibonacci number. Before returning this to the Python program, it converted into Python type by using the function Py_BuildValue(). These steps are similar for all the C functions being used.

Declaration of Methods

ModMethods[ ] is a list of all the methods implemented. Each entry in the list has the name of the method that can be called form Python followed by the name of the C function implementing it. In the example aforementioned, `fib` is the Python name given for `find_fib`, the actual C Fibonacci function. Each entry also has a flag describing the way in which the implementation function is called and also a brief description for the method. The last entry in the list `ModMethods[ ]` should always be a null entry.

Initializing the Extension Module

The function `initFibonacci( )` initializes the extension module. When the Python program imports the module Fibonacci, the first function to be called is `initFibonacci( )`. `Py_InitModule( )` creates a "module object" and inserts built-in function objects into the newly created module based upon the list that was passed as its second argument. `Py_InitModule( )` returns a pointer to the module object that it creates [Python].

After a clear understanding of the above tools, one can know the importance of a shared library which really helps in loading functions declared externally in another foreign programming language.

8. Limitations

Since Unicon string semantics are not the same as C strings, whenever a call is made to the C library function such as `strcat(s1, s2)` which concatenates two strings passed and holds the resultant string in s1, problems arise. As the C runtime system allocates enough new space for the string `s1` to accommodate the string `s2` which is not possible in Unicon because the Unicon string literals are Immutable types meaning one cannot modify the value of a string variable. This causes the Unicon runtime system to generate memory violation errors. Also there is no clear solution as to how to handle this problem efficiently. Only integer and real (double) single dimensional array types are handled in the interface. C language has many types for which Unicon has no equivalent data types and currently the support is provided for a few C data types.

9. Conclusion and Future Work

With the implementation of a new interface for Unicon language, it became easier to call externally available C functions inside Unicon. This new feature supports data types such as integers, real, characters and single dimensional array types. A number of C programs/functions already written and used can now be called inside Unicon code with the help of this new interface. There is no need for the programmer to write their own wrapper code or worry about building the shared library and loading the external C functions at run time. All this is neatly taken care of.

Even though a lot of work has been done in developing and improving this interface, there are a few more issues to be handled: support for even complex data types such as C

structures need to be provided. Making Unicon flexible enough to handle foreign language code will be future work. The immediate issues that lie ahead as a continuation to this work are:

- Capability of inlining the C functions with changes to `loadfunc()` for loading the external C functions directly without any interface.
- Making it possible to provide the `.o` or object files from which the external C functions are being loaded inside Unicon, on the command line while compiling a Unicon program.
- The code of external C function(s) which are being called should be directly embedded or linked into the Unicon Virtual Machine instead of generating wrapper code and corresponding Unicon/Icon stubs for calling the C functions.
- Integrating the macros used for C wrapper code, inside the Unicon language system to stop including the "`icall.h`" when writing C wrapper files.
- Extending the support for multi-dimensional arrays, different data structures such as structures (struct *), union and many more.
- Optimizing the performance of macros currently in use for handling lists.

10. Acknowledgements

The new preprocessor directives are basically the work of Sudarshan Gaikaiwari. This research was supported in part by an appointment to the NLM Research Participation Program. This program is administered by the Oak Ridge Institute for Science and Education for the National Library of Medicine.

Appendix A: Interface Macros Used for Wrapper Code Generation

(a) Integer Macros

ArgInteger(i) - this macro checks whether the argument `argv[i]` is an integer and returns an error if not, saying integer expected.

IntegerVal(d) - this macro does the real conversion of argument 'i' from Unicon type into a C integer type.

RetInteger(i) - this macro finally returns the resulting C integer value back to Unicon.

(b) Real Macros

ArgReal(i) - this macro checks whether the argument `argv[i]` is of double(real) type and returns an error if not, saying double/real expected.

RealVal(d) - this macro does the real conversion of argument 'i' from Unicon type into a C integer type.

RetReal(v) - this macro finally returns the resulting C real(double) value back to Unicon.

(c) Character Macros

ArgString(i) - this macro checks whether the argument `argv[i]` is a string and returns an error if not, saying character/string expected.

StringVal(d) - this macro converts the value passed in into a C char pointer and adds a '\0' for termination if necessary.

The following are the macros helpful in returning back the C strings (character pointer) values back to Unicon.

RetString(s) - returns a null terminated string s
RetStringN(s, n) - return string 's' whose length is 'n'
RetAlcString(s, n) - return already-allocated string
RetConstString(s) - return constant string s
RetConstStringN(s, n) - return constant string s of length n

(More descriptions regarding these macros can be found in the form of comments in the "`icall.h`" header file.)

(d) Lists (Arrays) Macros

ArgList(i) - this macro checks whether the argument `argv[i]` is an Icon list descriptor or not and returns an error if not.

IListVal(d, a) - converts the Icon list descriptor 'd' of integers into an equivalent C array 'a' of integers.

RListVal(d, a) - converts the Icon list descriptor 'd' of real's(doubles) into an equivalent C array 'a' of doubles.

ListLen(d) - the length of the Icon list descriptor passed can be found with this macro.

The following listed macros are helpful in converting C integer/real arrays into corresponding Icon/Unicon lists and are then returned back to Unicon.

mkIlist(int x[], int n) - this make integer list macro takes a C array and a variable having the value of its size computed from the aforementioned ListLen macro. Resultantly creates a Unicon list descriptor of integer values for passing back.

mkRlist(int x[], int n) - this make real(doubles) list macro takes a C array and a variable having the value of its size computed from the aforementioned ListLen macro. Resultantly creates a Unicon list descriptor of real (double) values for passing back.

11. References

[Jeffery03] Clinton, L. Jeffery, S. Mohamed, R. Pereda, and R. Parlett. *Programming with Unicon*. http://unicon.sourceforge.net/book/ub.pdf, 2003.

[Jeffery02] Clinton, L. Jeffery and Shamim Mohamed. *Unicon Language Reference*. Unicon Technical Report #8. Aug 2002.

[IA36] Ralph E. Griswold, Madge T. Griswold, and Gregg M. Towsend – *The Icon Analyst* Volume 36, June 1996.

[IB02] Ralph E. Griswold, and Madge T. Griswold. *The Icon Programming Language*. 3rd ed. 2002.

[IU04] Ralph E. Griswold, Madge T. Griswold, Kenneth W. Walker, and Clinton L. Jeffery. *The Implementation of Icon and Unicon* 2004. This is an unpublished document.

[Liang] Sheng Liang - *Java Native Interface: Programmer's Guide and Specification* Addison Wesley, June 1999.

[SWIG] SWIG Tutorial. http://www.swig.org

[MAN] UNIX Man pages

[Wiki] *Wikipedia, the free encyclopedia*. http://en.wikipedia.org.

[UA] Mixing Python and C. http://rlai.cs.ualberta.ca/RLAI/RLsoftware/PythonC.html

[Python] Extending and Embedding the Python Interpreter. http://docs.python.org/ext/methodTable.html, Sept 2005.