Simplified Wrapper and Interface Generator

David M. Beazley

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
beazley@cs.utah.edu

- and -

Theoretical Division
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
beazley@lanl.gov

February 22, 1996

Version 0.1 Alpha

Version 0.1 Alpha, February 1996

1. Introduction

What is SWIG?

Simply stated, SWIG is a simple tool I have developed to make it more enjoyable for scientists and programmers to integrate common interface languages such as Tcl, Perl, and Guile with programs containing collections of functions written in C or C++. Most interface languages provide mechanisms for accessing C variables and functions, but this almost always requires the user to write highly specialized wrapper code. While this may not be difficult, it is almost always tedious. SWIG automates this process and makes it extremely easy for a user to extend their favorite interface language without having to worry about alot of grungy details. This has a number of useful applications including :

- Simplified user-interface programming.
- Extremely rapid prototyping.
- Improved debugging and testing.
- Better performance (by making it easy for a user to write functions in C instead of Tcl or Perl)
- Development of language independent applications.

I originally developed SWIG to serve the needs of computational physicists. Having worked in this field for several years, I was frustrated by the difficulty of creating flexible user interfaces for scientific applications and the enormous amount of time that was wasted trying to do so. I wanted to develop a system that would make it extremely easy for scientists to put together interesting applications involving numerical simulation, data analysis, and visualization without having to worry about tedious systems programming or making substantial modifications to existing code.

The history of SWIG

I developed the first SWIG prototype in July, 1995 when I was working on my own interface language for controlling molecular dynamics (MD) simulations running on massively parallel supercomputers. In reality, SWIG was developed out of laziness-- I was faced with the problem of writing interface wrappers for more than 200 functions from our MD code and I thought that it would be a lot easier to write an automated tool instead. Later, I realized that this approach could work nicely with other types of languages such as Tcl, Perl, and Guile.

Simplicity vs. Functionality

This tool was originally developed to make it easy to put interesting interfaces on existing applications. My primary goal was to minimize user-involvement as much as possible---especially since the scientific users wanted to focus on scientific issues instead of systems programming. Therefore, you may find its capabilities rather limited at times---in fact, SWIG may only use a small subset of each interface language. However, I have always felt that simplicity and functionality must be balanced (after all, what good is it if the easy to use tool is more complicated than the original problem?). With that said, SWIG will certainly not do everything. However, I have found it to be remarkably powerful in my own work and I hope that you find it to be just as useful.

How do I get SWIG?

The official location of SWIG related material is

```
http://www.cs.utah.edu/~beazley/SWIG/swig.html
```

This site contains the latest version of the software, users guide, and information regarding bugs, installation problems, and implementation tricks.

Acknowledgements

This work would certainly not be possible without the support and patience of many people. I would like to acknowledge Peter Lomdahl, Brad Holian, Shujia Zhou, Niels Jensen, and Tim Germann at Los Alamos National Laboratory for allowing me to pursue this project and for being the first users. Patrick Tullmann at the University of Utah suggested the idea of automatic documentation generation which was a cool solution to a problem I had been thinking about for awhile. John Schmidt and Kurtis Bleeker at the University of Utah tested out the most recent versions. I'd also like to acknowledge Chris Johnson and the Scientific Computing and Imaging Group at the University of Utah for their continued support. Finally, I'd like to acknowledge Jeff Johnson at Los Alamos National Laboratory for always having the answer to the strangest of computing questions and listening to all of my crazy ideas over the years.

2. An Overview

Figure 1 shows the structure of SWIG. Input is specified using a “interface file” containing ANSI C style declarations of functions and variables. This file is parsed and passed on to a code-generator and a documentation module. The code generator consists of a collection of modules containing the variable and function linkage rules of different interface languages. Similarly, the documentation system can produce different types of output formats. The code generator and documentation modules are independent, yet documentation is produced for whatever interface language is being used (for example, if using Perl, the documentation will use Perl syntax).

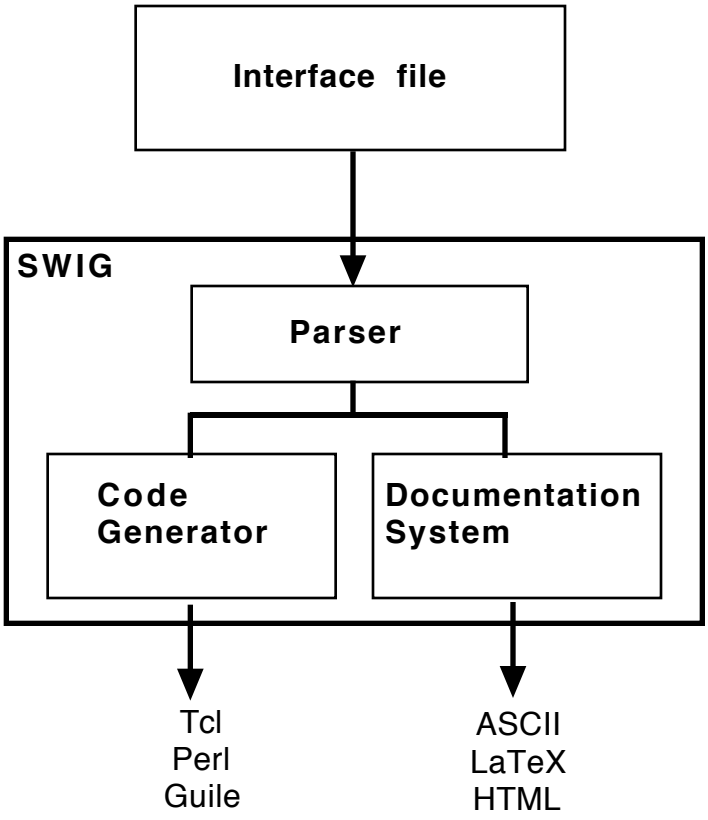


Figure 1. SWIG Organization

3. A Simple Example

Let's start with a simple example. Consider the following C code :

```
/* File : example.c */

double  My_variable  = 3.0;

int  fact(int n) {
     if (n <= 1) return 1;
     else return n*fact(n-1);
}

int mod(int n, int m) {
     return(n % m);
}
```

Now suppose that you wanted to add these functions and the global variable `My_variable` to the `tclsh` program. We start by making a SWIG "interface" file as shown below (by convention, these files carry a `.i` suffix) :

```
/* File : example.i */

%{
/* Put headers and other declarations here */
%}

%include tclsh      // Link with tclsh

extern double My_variable;
extern int    fact(int);
extern int    mod(int n, int m);
```

The interface file contains ANSI C function prototypes and variable declarations. In addition, the file contains the directive "`%include tclsh`". This inserts code for the `Tcl_AppInit()` function that `tclsh` needs in order to initialize user defined functions.

SWIG is invoked using the `wrap` command. We can use this to build our new `tclsh` program and run it as follows :

```
% wrap -tcl example.i
Generating wrappers for Tcl.
Documentation written to example_wrap.tex
% gcc example.c example_wrap.c -ltcl -lm -o my_tclsh
# Run it
% my_tclsh
% fact 4
24
% mod 23 7
2
% expr $My_variable + 4.5
7.5
%
```

SWIG produced a new file called `example_wrap.c` that can be compiled and linked with the `example.c` file and the Tcl library. `my_tclsh` is functionally identical to the original `tclsh` program except that it now has our variables and functions added to it. The file `example_wrap.tex` contains documentation, but this will be discussed a little later.

Now, let's add the same functions to Perl4. Making no code modifications, we do the following :

```
% wrap -perl4 example.i
Generating wrappers for Perl4
Looking up tclsh in lib/perl4 ... not found
Documentation written to example_wrap.tex
% gcc example.c example_wrap.c uperl.o -lm -o my_perl
# Run it
% my_perl
print &fact(4), "\n";
print &mod(23,7), "\n";
print $My_variable + 4.5, "\n";
<ctrl-d>
24
2
7.5
%
```

In a manner of minutes, we have been able to add our C functions to both Tcl and Perl without writing any special code. This is only one example of the flexibility of SWIG.

4. Wrappers and interface languages

SWIG eliminates most of the grungy details of extending interface languages, but it is important to understand what's going on inside. Most interface languages provide some mechanism for adding new C functions and variables, but this usually requires one to write special language-dependent wrapper code. To see what happened in our example, we can take a look at the Tcl implementation of `example_wrap.c`. Much of the code follows the style described in "Tcl and The Tk Toolkit."

```
/*
 * File : example_wrap.c
 * Thu Feb  8 01:29:37 1996
 *
 * Simplified Wrapper and Interface Generator (SWIG)
 *
 * Copyright (c) 1995,1996
 * The Regents of the University of California and
 * The University of Utah
 *
 */

/* Implementation : TCL */

#define INCLUDE_TCL    <tcl.h>
#define INCLUDE_TK     <tk.h>
#include INCLUDE_TCL
#include <string.h>
#include <stdlib.h>

/* Put headers and other declarations here */
#define WG_init    init_wrap

extern double My_variable;
extern int fact(int );
extern int mod(int ,int );

/* A TCL_AppInit() function that lets you build a new copy
 * of tclsh.
 *
 * The macro WG_init contains the name of the initialization
 * function in the wrapper file.
 */

#ifndef WG_RcFileName
char *WG_RcFileName = "~/.myapprc";
#endif
```

```
int main(int argc, char **argv) {
  Tcl_Main(argc, argv, Tcl_AppInit);
  return(0);
}
int Tcl_AppInit(Tcl_Interp *interp){
  int WG_init(Tcl_Interp *);
  if (Tcl_Init(interp) == TCL_ERROR)
    return TCL_ERROR;
 /* Now initialize our functions */
  if (WG_init(interp) == TCL_ERROR)
    return TCL_ERROR;
  tcl_RcFileName = WG_RcFileName;
  return TCL_OK;
}
int _wrap_fact(ClientData clientData, Tcl_Interp *interp,
               int argc, char *argv[]) {
 int _result;
 int _arg0;
 if (argc != 2) {
       interp->result = "wrong # args";
       return TCL_ERROR;
}
 _arg0 = atoi(argv[1]);
 _result = fact(_arg0);
 sprintf(interp->result,"%ld", (long) _result);
 return TCL_OK;
}
int _wrap_mod(ClientData clientData, Tcl_Interp *interp,
              int argc, char *argv[]) {
 int _result;
 int _arg0;
 int _arg1;
 if (argc != 3) {
       interp->result = "wrong # args";
       return TCL_ERROR;
}
 _arg0 = atoi(argv[1]);
 _arg1 = atoi(argv[2]);
 _result = mod(_arg0,_arg1);
 sprintf(interp->result,"%ld", (long) _result);
 return TCL_OK;
}

int init_wrap(Tcl_Interp *_wrap_interp) {
 if (_wrap_interp == 0)
       return TCL_ERROR;
 Tcl_LinkVar(_wrap_interp, "My_variable",
             (char *) &My_variable, TCL_LINK_DOUBLE);
```

```
    Tcl_CreateCommand(_wrap_interp, "fact", _wrap_fact,
                      (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
    Tcl_CreateCommand(_wrap_interp, "mod", _wrap_mod,
                      (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
    return TCL_OK;
}
```

Thus, an interface file containing only 3 declarations and 1 directive has turned into approximately 100 lines of wrapper code for Tcl. In practice, SWIG can generate several thousands of lines of code corresponding to several hundred functions. Fortunately, it is rarely necessary to look at this code, but you could use it as a starting point for developing more sophisticated packages.

5. Interface files

As input, SWIG takes a an interface file with the following format :

```
%extern            // Optional. Used for multiple files
%init my_init      // Optional. Name of initialization function
%{
      // Include header files and other stuff here
      // Everything is copied directly into the output file
%}

// Now list variable and function declarations (in any order)
```

The start of the file contains a few optional directives that can only be placed at the beginning of the file. Between the `%{,%}` you can place anything you like---everything in this section is copied verbatim into the resulting wrapper file. This section must always be included even if it is empty. After the header information, simply list declarations in ANSI C syntax in any order that you like.

By default an interface file with the name `myfile.i` will be transformed into a file called `myfile_wrap.c` that should be compiled and linked with the rest of the program.

C and C++ style comments may be placed in interface files, but these are used to support the automatic documentation system. Please see the documentation section for more details on this. Otherwise, SWIG throws out all comments so even if you use a C++ style comment, the resulting wrapper file can still be compiled with the C compiler.

6. Syntax

SWIG's syntax is almost exactly the same as that used in C header files to declare variables and functions. However, It was not not my intent to write a full C compiler. Therefore, there are a few basic rules to follow.

- Use ANSI C prototypes.
- Comments may be safely placed anywhere and should be enclosed in `/*,*/` or proceeded by `//`
- SWIG directives are always preceeded by a "%"
- C preprocessor directives may be inserted anywhere, but the result may not be what you expect. SWIG does not run CPP at any stage of its processing. As a general rule the C preprocessor can really only be safely used for conditional compilation.
- Only use pointers to complex data-types (structs, classes, etc...).
- Do not assign values to variables.

As a general rule, SWIG understands simple C declarations of variables of functions. If you are in doubt whether something will work, try it and see--SWIG will complain if you give it something it doesn't understand. As long as you keep in mind that interface files are not header files, you should be have very few problems. Of course, if you do decide to try something tricky with the CPP, don't say I didn't warn you!

7. Declarations

7.1. Syntax

Again, all declarations must follow ANSI C syntax. At the present time, support for C++ declarations is limited although work is in progress. The following list shows some valid declarations :

```
          int      a;
         double    x,y,z;
extern   char     *message;
extern   int       foo(int a, int b);
extern   double   *make_array(int);
         void      print_data(double **);
unsigned int       f1(void), f2();
extern   MyType   *bar(int);
extern   int       MyClass::foo(void)
```

Each variable or function must have an associated type. There are no implied types (such as integers). Secondly, most C datatypes are supported in addition to complex user-defined types (as discussed in a moment). Finally, lists of variables of the same type are permitted such as in "`double x,y,z;`".

With that said, here are some invalid declarations :

```
extern  myvar;              // Error. no specified type
int foo(a,b)                // Do not use old C style
     int a,b;
char  my_string[50];        // Don't do this--trust me.
void  func(struct node *a) // Don't use "struct". Can fix
                            // easily with a typedef however.
void  bar(int &a, int &b)  // C++ not fully supported yet
```

Like with header files, the extern keyword should be attached to any declaration that hasn't been previously declared in the interface file.

7.2. What's in a name?

Variable and function names must use the same rules as for C identifiers. However, SWIG also allows the characters `{- > . :}` to be part of identitifers. What does this mean? Primarily, it means that the following are all legal SWIG declarations :

```
extern   int   v.x;             // Variable linkage to member
extern   double MyClass::foo()  // Static class member functions
extern   double M->bar()        // Member functions
```

Of course, as you might imagine, this is also a really great way to shoot yourself in the foot. However, it can lead to some interesting tricks too.

7.2. Data Types

Interface languages are usually quite limited in their support for typed data. For example, Tcl really only works with strings although it can link to C integer, double, or character string data. Trying to accommodate all possible data types and different interface languages would be a difficult task so SWIG takes a compromise approach.

Internally, SWIG is capable of supporting the following data types :

```
int, short, long, float, double, char, void, user_defined
```

The `user_defined` type can be any identifier used in a type position. For example, if you declare :

```
int  foo(Matrix *a);
```

SWIG will assume that `Matrix` is some sort of user-defined type that has been defined somewhere.

Pointers may be used with any of the above types. In addition, the modifiers `unsigned` and `signed` may also be used. The above types should be adequate for most purposes. However, the use of certain types may be restricted in certain languages. A warning or error message will be generated if this is the case.

You can also use the `%typedef` directive to map certain datatypes to any one of the SWIG basic datatypes. For example :

```
%typedef unsigned int size_t
```

will make `size_t` behave exactly like an unsigned integer. Otherwise SWIG would treat it as some unknown user-defined type.

7.3. Variable Linkage

The process of variable linkage allows an interface language to directly manipulate some global variable in a program. For example, the result of a Tcl expression could be placed directly into some global variable. Unfortunately, the extent to which different languages can link variables varies widely and it is almost impossible to accomodate all possibilities. Furthermore, some care must be taken to avoid clobbering yourself (since having a language directly manipulate one of your variables may be unsafe--or least untrusted). SWIG allows linking, but only according to the following rules:

- Whenever possible, a variable will be linked with the language, but only if it can be done safely. A linkage is "unsafe" if it would result in a type-mismatch or if a result was larger than the space allocated for it (for example, trying to store a 64 bit long int into a 32 integer would be unsafe).

- If linkage is not possible, functions to set and get the values of a variable may be created instead. For example :

  ```
  extern short  a;
  ```

  may get translated into the following functions which get added to the interface.

  ```
  void set_a(short value);
  short get_a();
  ```

- Otherwise, a warning message will be displayed and the declaration ignored.

As a general rule, the basic data types of `int`,`double`, and `char *` can always be linked. Support for other types vary widely. Support for pointers (other than char *) is even more limited. Finally, no linkage is currently provided for any user-defined types such as structs or classes (although you can sometimes link to class member variables and functions if you are careful).

7.4. A word about character strings

Linking character strings presents a special problem to most interface languages. Consider for a moment the following variable :

```
char   my_string[64];
```

Now suppose that this variable was linked to Tcl and the user decided to change it. If the new string was less than 64 characters, then there are no problems, but once the result exceeded this size, the code would either generate a segmentation fault or overwrite some other part of memory. In either case, this is undesirable.

To avoid this problem, many interface languages will automatically free and allocate space for string results. Thus, a safe implementation would free the memory used by `my_string`, and allocate a new block of memory with enough space for the result.

The dynamic treatment of strings in most interface languages, makes the string declaration above unsafe in most cases. Therefore, it is highly recommended that all strings be declared as pointers as in

```
char  *my_string;
```

If necessary, this string could be initialized using

```
my_string = (char *) malloc(64);
```

Whatever the case, this latter implementation should be used whenever possible. A related problem with strings is the use of character values. Consider for a moment the declaration :

```
void foo(char a);
```

Does "`a`" correspond to a character entered as a string or is it some sort of 8-bit numeric value? Similiarly , consider :

```
void foo(char *a);
```

Is “a” a string or is a pointer to some sort of character buffer? To resolve these issues you can do one of the following :

```
void foo(char a);          // a is an ASCII character
void foo(signed char a);   // a is an 8-bit signed integer
void foo(unsigned char a); // a is an 8-bit unsigned integer
void foo(char *a);         // a is a character string
void foo(signed char *a);  // a is a pointer
void foo(unsigned char *a);// a is a pointer
void foo(void *a);         // a is a pointer
```

7.5. Functions

Support for functions is considerably more flexible than for variables. Functions may return any of the basic types (including pointers) and take arguments involving any of the datatypes listed earlier. This increased flexibility is mainly due to the fact that all of these datatypes can be accurately represented in each interface language (even if it’s in the form of a string). User defined types (such as structs and classes) may also be used, but functions may only return pointers to complex data-types.

7.6. Pointers and user defined types

SWIG supports pointers and pointers to user-defined types such as structs and classes. The representation of pointers in SWIG is language-dependent, but at the very least, pointers can be aded to all interface languages by representing them as a character string. Some languages may provide pointer support, but even in this case, the SWIG representation may be different. Thus, some care some be taken to only use pointers created by SWIG with other SWIG-generated functions.

There are a few restrictions on the declaration of functions involving user-defined types as shown below :

```
Vector *cross(Vector *, Vector *); // This is okay
```

Compare with the following declaration which is illegal.

```
Vector cross(Vector *, Vector *); // This is illegal
                                  // Can’t return by value.
```

SWIG can also pass complex types around by value as shown :

```
Vector *cross(Vector, Vector);      // This is okay
```

Having just said that SWIG uses pointers for everything, this last example may seem strange. However, my thinking here is that a user may really want to use a function that takes arguments by value. Provided the user creates some objects and has pointers to those objects, then one should be able to call the function as

```
Vector *_wrap_cross(Vector *a, Vector *b) {
     return cross(*a, *b);
}
```

SWIG automatically generates code to do this, but since this could be unsafe, a warning message will be printed.

Note : I've thought about allowing functions to return complex datatypes by value as well. This could be done by creating a new object, assigning the returned function value to it, and returning a pointer to the new object. The user would be responsible for freeing the memory however--surely a potential source of memory leaks. This may or may not be supported in a later release.

7.7. Read-only variables

Sometimes it is desirable to allow a user to access, but not modify certain global variables. This can be done using the `%readonly` and `%readwrite` directives as shown in the following example :

```
%{
%}

int  a;           // can read/write
%readonly
int  b,c,d;       // Read only variables
%readwrite
double x,y;       // read/write
```

The `%readonly` directive enables read-only mode until it is explicitly disabled using the `%readwrite` directive.

7.8. Resolving name conflicts

Normally, SWIG takes each declaration and adds it to the interface language using the same name. Unfortunately, this sometimes creates a naming conflict with some command or other variable that is already part of the language. Variables and functions can be “renamed” by using the `%name` directive as shown below :

```
%{
%}

%name my_print {extern void print(char *); }
%name foo {extern int a_really_long_and_annoying_name;}
```

SWIG still links to the proper variables and functions, but `%name` simply lets you rename something within the interface language. In this case, the `print()` function will really be called `my_print()` in the interface. You can also use this feature to simply change the name of variables or functions to something simple and or friendly if you want.

7.9. Overriding parameter passing rules

By default, SWIG attempts to produce wrapper code that exactly matches the original C function. However, suppose you wanted to use some function written in Fortran such as the following from the Linpack library :

```
      subroutine dgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      double precision a(lda,1)
c
c     dgefa factors a double precision matrix by gaussian elimination.
c
c     (Dave's note: I stripped the comments down a bit here)
c     Inputs : a, lda, n
c     Outputs : a, ipvt, info
c
c     linpack. this version dated 08/14/78 .
c     cleve moler, university of new mexico, argonne national lab.
c
```

SWIG can handle such a function, but it will have to rely on the linker to combine C and Fortran object files (which varies widely between machines). In the interface file we might specify the following :

```
extern void dgefa_(double *, int *, int *, int  *, int *);
```

Assuming that you can get this to compile, you could call this function through Tcl just like any other function. However, Fortran expects all of its arguments to be passed by reference and SWIG interprets this quite literally---expecting pointers for all 5 arguments. You can change this by using the `%val` directive as follows :

```
extern void dgefa_(double *, %val int *, %val int *,
                   int *, %val int *);
```

Now, arguments 2,3 and 5 can be passed by value from Tcl or Perl as shown :

```
% dgefa_ $matrix 10 10 $pivots 0
```

Internally, the `%val` directive forces a temporary variable to be created. The argument value is stored in this variable and a function call is made using a reference to it.

8. Multiple Files

It is often desirable to use multiple interface files for increased modularity and flexibility. SWIG supports this, but it is necessary to indicate this in the interface file since SWIG sometimes declares global variables (which would end up being duplicated). The following example shows how to use multiple files :

```
/* File : interface.i    */

%extern                              // Link to external module
%init interface_init, user_init      // call user_init()
%{
#include "my_header.h"
%}

// Declare some common functions

double  *read_data(char *filename);
void    save_data(char *filename, double *a);
void    print_data(double *a);

....

/* File : user.i */

%init user_init
%{
#include "my_header.h"
%}

// User defined functions

extern void my_func(double *d);

....
```

Here we have two files, `interface.i` and `user.i`. The first file, `interface.i`, specifies that it is going to externally link to global internal variables declared in another wrapper file. Furthermore, in the `%init` line, we specify the name of the initialization function as `interface_init()` , but we also specify the name of another initialization function `user_init()` which also gets called.

This arrangement may seem strange at first, but it actually makes it possible to chain together a whole bunch of interface files. Perhaps the file `interface.i` contains some intrinsic functions in a software package while the file `user.i` contains user-

definable extensions. Each user could write their own version of `user.i`, but still link with the functions and variables declared in `interface.i`. This chaining process can even be extended further by the user. For example :

```
/* File : my_interface.i */

%init my_init
%{
#include "my_header.h"
%}

....

/* File : user.i */

%extern
%init user_init, my_init
%{
#include "my_header.h"
%}
....

/* File : interface.i */

%extern
%init interface_init, user_init
%{
#include "my_header.h"
%}

....
```

Here three interface files are chained together. Initialization is still easy---within the original software package, we make a call to `interface_init()`, but this in turn calls `user_init()`, which calls `my_init()`. Thus, one initialization call in the C code can be used to initialize functions from an almost arbitrary number of interface files.

It should be noted, that chaining is optional. For example, the following two files could be used :

```
/* File : interface.i */

%init interface_init
%{
#include "my_header.h"
%}
....

/* File : user.i */

%extern
%init user_init
%{
#include "my_header."
%}

....
```

The `%extern` directive is still needed to avoid naming conflicts, but the chaining is gone. In order to initialize the interface, one would have to execute the equivalent of the following code.

```
// Initialize the interface

interface_init();
user_init();
```

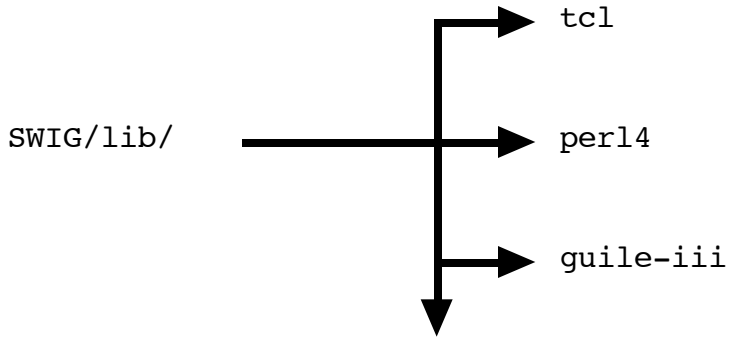
However, this may present certain problems for some languages. For example, Perl4 uses it's own `main()` function which makes a single call to `userinit()`. Thus, it may not be possible to make multiple initialization calls.

Note : I may rethink this whole mechanism of multiple file linkage. For now it works, but sometimes I find it a little weird.

9. Using Libraries

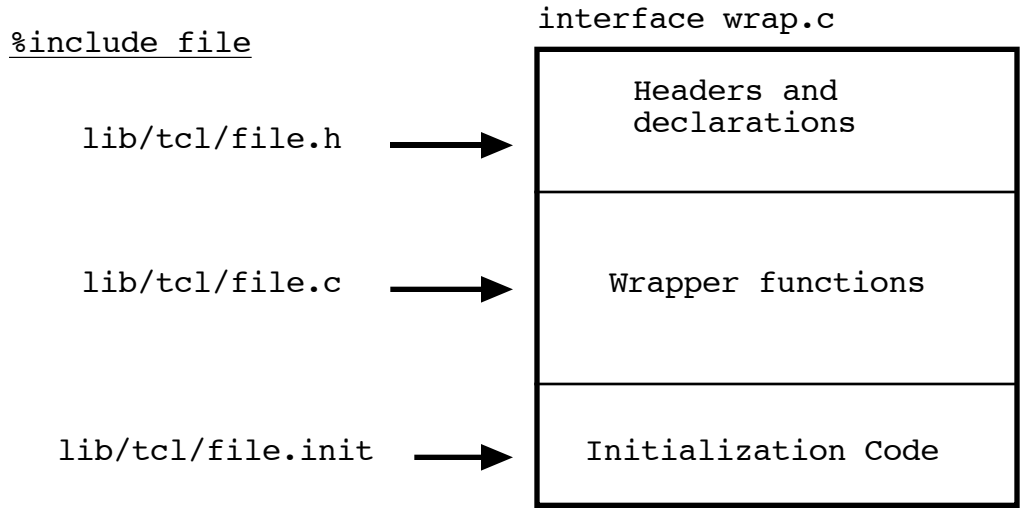
To allow greater flexibility, SWIG allows the user to build up a library of interesting functions that may or may not be language dependent. To include additional code, the `%include` directive should be used. We already saw an example of this earlier when we built a `tclsh` application.

All of SWIG's libraries are included in the `lib` directory of the SWIG distribution. This directory is organized as follows :



Within the `lib` directory, there is a directory for each interface language. Within each of these directories are the language dependent files.

The `%include` directive searches for three separate files that are inserted into the resulting wrapper file as shown below :



The `file.h` file should declare any include files and other necessary information. `file.c` should contain the actual C code and `file.init` should contain any initialization code that is needed. To illustrate how this works, consider the following code for building wish :

```
/* File : lib/tcl/wish.h */

#include <tk.h>

--------
/* File : lib/tcl/wish.c */

/* Need to supply main() for Tk4.0 */

int main(int argc, char **argv) {

  Tk_Main(argc, argv, Tcl_AppInit);
  return(0);

}

/* Initialize new commands */

int Tcl_AppInit(Tcl_Interp *interp)
{
    Tk_Window main;
    main = Tk_MainWindow(interp);

    if (Tcl_Init(interp) == TCL_ERROR) {
     return TCL_ERROR;
    }

    if (Tk_Init(interp) == TCL_ERROR) {
     return TCL_ERROR;
    }

    if (WG_init(interp) == TCL_ERROR) {
     return TCL_ERROR;
    }
    return TCL_OK;
}

---------
/* File : lib/tcl/wish.init */

< empty >
```

You can also create your own library directory of interesting function. It should have the same subdirectory structure as the SWIG library directory. To use it, you should run SWIG with the option `-I/my_path/my_lib` which forces SWIG to look in the specified library file in addition to its own libraries. SWIG will always search your path first making it possible to customize the files in the original library directory.

Unfortunately, library programming is still a little obscure. The real benefit is that one can develop a library of interesting code that can be reused in many different applications. It's also possible to write a different version of some module for each interface language. This can make it possible to easily build applications under multiple interface languages.

The `lib` directory contains some examples so look there for more information and pointers to what's currently available.

10. main() and name conflicts

SWIG makes it very easy to combine bits and pieces of different programs with various interface languages and other packages. This can sometimes create nasty global namespace conflicts. I know of no easy solution to this, other than to suggest to the user that they be careful. One approach may simply be to add an application specific prefix to all of your vaiables and functions.

A second problem revolves around the proper use of the `main()` function. Some interface languages provide their own `main()`, but others do not. If your program contains it's own version of `main()`, you may need to rename it or change it's implementation so that SWIG can be used effectively. Unfortunately, this seems to be highly specific to each application. In my own work, I tend to take the following strategy :

1. Do not write a `main()` function.
2. Create a special `init()` function to initialize any global variables and do other things that might have been done in a `main()` function had I written one. Declare `init()` in the interface file.
3. Let SWIG provide a main function (usually by including tclsh, wish, or some other package).
4. Put a call to `init()` in any input scripts if necessary.

11. The documentation system

Just as SWIG can produce wrapper code, it also can produce documentation describing the user-interface that has been constructed. While this is certainly no substitute for a full-blown documentation system, it is handy to know what it was you just added to your favorite interface language---and it beats a kick to the head.

SWIG keeps an internal record of every variable and function that has been declared in the interface file. At the end of processing, these entires are sorted, categorized, and dumped to a documentation file in any number of formats including ASCII, LaTeX, and HTML.

11.1. Documentation command line options

The type of documentation is selected using the following command line options :

- `wrap -dascii  # Produce ASCII doc`
- `wrap -dlatex  # Produce LaTeX`
- `wrap -dhtml  # Produce HTML`

11.2. Adding function argument names

By default, the documentation system only produces a list of variable and function names along with the types of function arguments. You can provide more information by declaring functions with argument names. For example :

```
void  read_matrix(char *filename, Matrix *matrix);
```

SWIG ignores the names when producing C code, but they are passed to the documentation system. When documentation is produced, the `read_matrix` function will list its arguments as `filename` and `matrix` which is alot more descriptive than simply printing the datatypes (which may be fairly nondescriptive in most cases).

11.3. Adding descriptions

Descriptions may also be attached to each variable or function by entering a C/C++ style comment immediately after its declaration in the interface file. For example :

```
void  read_matrix(char *filename, Matrix *matrix);
/* Reads a matrix from disk.  filename specifies the input
   file */
```

This would produce a documentation entry that not only gave the function declaration, but also a short description describing it's use.

11.4. Titles and categories

Interface files may also be given titles and broken up into categories. To use a title, place a `%title` directive at the very beginning of an interface file. To break up commands into sections, place the `%section` directive into the interface file. For example :

```
%title "Example file"
%init my_init
%{
#include "my_header.h"
%}

%section "Mathematical operations"

... declarations ...

%section "Graphics"

... more declarations ...

%section "File I/O"

...more declarations...
```

When the final documentation is printed, it will be divided into categories, alphabetized, and broken down by variables and functions. Sections are divided according to the section name provided. Thus, if you later gave a `%section "Graphics"` in the above file, those declarations would be grouped with the first set.

11.5. Formatting

The documentation system tries to produce readable output. This can be greatly enhanced by adding your own formatting commands. For example, in LaTeX mode, you can simply place embedded LaTeX commands in the C comments. Since both LaTeX and HTML do their own formatting not much else is required. For ASCII mode, SWIG strips all white space and line-breaks and attempts to format the comments to fit nicely on an 80 column print-out. However, the special character sequence `"\\"` will force the ASCII mode to produce a line break.

12. A More Complex Example

Now let's consider a more complicated example. In this example, we'll consider code for manipulating 4x4 transformation matrices used for 3D graphics and illustrate how C code can be integrated with Tcl.

Starting C code :

```
/*   FILE : matrix.c : some simple 4x4 matrix operations */

/* Create a new matrix */

double **new_matrix() {
  int i;
  double **M;

  M = (double **) malloc(4*sizeof(double *));
  M[0] = (double *) malloc(16*sizeof(double));
  for (i = 0; i < 4; i++)
    M[i] = M[0] + 4*i;
  return M;
}

/* Destroy a matrix */

void destroy_matrix(double **M) {
  free(M[0]);
  free(M);
}

/* Print out matrix (for debugging) */

void print_matrix(double **M) {
  int i,j;
  for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++)
      printf("%10g ", M[i][j]);
    printf("\n");
  }
}

/* Matrix multiply.  m1*m2 -> m3 */

void mat_mult(double **m1, double **m2, double **m3) {
  int i,j,k;
  double temp[4][4];
```

```
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) {
      temp[i][j] = 0;
      for (k = 0; k < 4; k++)
         temp[i][j] += m1[i][k]*m2[k][j];
    }
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++)
      m3[i][j] = temp[i][j];
}
```

So far we have defined a few operations, but we also need some way to access data in each matrix. Interface languages don't know about matrices so we can easily provide two new functions designed specifically for the user-interface :

```
/* Set matrix element (i,j) */

void set_m(double **M, int i, int j, double val) {
  M[i][j] = val;
}

/* Get matrix element (i,j) */

double get_m(double **M, int i, int j) {
  return M[i][j];
}
```

Now let's write the interface file :

```
/*  FILE  :  matrix.i  */

%init matrix_init
%{
#include <math.h>
%}
extern double **new_matrix();
extern void destroy_matrix(double **);
extern void print_matrix(double **);
extern void set_m(double **, int, int, double);
extern double get_m(double **, int, int);
extern void mat_mult(double **a, double **b, double **c);
```

We must now decide how we want to run our matrix program. As is, we have not specified a `main()` function or whether the program should be included in some other package such as `tclsh` or `wish`. For now, let's write our own `main()` :

```
/* File : main.c */

#include <tcl.h>
extern int matrix_init(Tcl_Interp *); /* Init function from matrix.i */

int main() {
  int        code;
  char       input[1024];
  Tcl_Interp *interp;

  interp = Tcl_Create_Interp();
  if (matrix_init(interp) == TCL_ERROR)
      exit(0);
  fprintf(stdout,"matrix > ");
  while(fgets(input, 1024, stdin) != NULL) {
    code = Tcl_Eval(interp, input);
    fprintf(stdout,"%s\n",interp->result);
    fprintf(stdout,"matrix > ");
  }
}
```

Compiling and running it :

```
% wrap -tcl matrix.i
% gcc matrix.c matrix_wrap.c main.c -ltcl -lm -o matrix
% matrix
matrix > set M [new_matrix]
100084e0
matrix > print_matrix $M
      0          0          0          0
      0          0          0          0
      0          0          0          0
      0          0          0          0

matrix > set_m $M 0 0 1
matrix > print_matrix $M
      1          0          0          0
      0          0          0          0
      0          0          0          0
      0          0          0          0

matrix > <ctrl-d>
%
```

So far this isn't too exciting, but let's write a Tcl script to implement some matrix operations (this probably wouldn't be too efficient, but it illustrates the idea).

```
# File : matrix.tcl
# Make a zero matrix

proc zero M {
    for {set i 0} {$i < 4} {incr i 1} {
        for {set j 0} {$j < 4} {incr j 1} {
            set_m $M $i $j 0.0
        }
    }
}

# Make an identity matrix

proc identity M {
    zero $M
    for {set i 0} {$i < 4 || $i < 4} {incr i 1} {
         set_m $M $i $i 1.0
    }
}

# Rotate around x axis r degrees

proc rotx {M r} {
    set temp [new_matrix]
    set rd [expr $r*3.14159/180]
    zero $temp
    set_m $temp 0 0 1.0
    set_m $temp 1 1 [expr cos($rd)]
    set_m $temp 1 2 [expr -sin($rd)]
    set_m $temp 2 1 [expr sin($rd)]
    set_m $temp 2 2 [expr cos($rd)]
    set_m $temp 3 3 1.0
    mat_mult $M $temp $M
}

# Rotate around y axis r degrees

proc roty {M r} {
    set temp [new_matrix]
    set rd [expr $r*3.14159/180]
    zero $temp
    set_m $temp 1 1 1.0
    set_m $temp 0 0 [expr cos($rd)]
    set_m $temp 0 2 [expr sin($rd)]
    set_m $temp 2 0 [expr -sin($rd)]
```

```
    set_m $temp 2 2 [expr cos($rd)]
    set_m $temp 3 3 1.0
    mat_mult $M $temp $M
}

# Rotate around z axis r degrees

proc rotz {M r} {
    set temp [new_matrix]
    set rd [expr $r*3.14159/180]
    zero $temp
    set_m $temp 0 0 [expr cos($rd)]
    set_m $temp 0 1 [expr -sin($rd)]
    set_m $temp 1 0 [expr sin($rd)]
    set_m $temp 1 1 [expr cos($rd)]
    set_m $temp 2 2 1.0
    set_m $temp 3 3 1.0
    mat_mult $M $temp $M
}

proc scale {M s} {
    set temp [new_matrix]
    zero $temp
    for {set i 0} {$i < 4} {incr i 1} {
         set_m $temp $i $i $s
    }
    mat_mult $M $temp $M
}
```

Now we run our matrix program again with our new Tcl script :

```
% matrix
matrix > source matrix.tcl
matrix > set M [new_matrix]
100086a8
matrix > identity $M
matrix > rotx $M 45
matrix > rotz $M 30
matrix > scale $M 0.5
matrix > print_matrix $M
  0.433013      -0.25          0          0
  0.176777   0.306187  -0.353553          0
  0.176776   0.306186   0.353554          0
         0          0          0        0.5

matrix > <ctrl -d>
%
```

Extension with Complex Data Type

From our simple example, we have integrated a 2-d matrix declared in C with Tcl and used it in a Tcl script involving both Tcl and C functions. More importantly, the C code looked like normal C code while the Tcl code looked like normal Tcl code.

Now let's extend our program to include vectors so that we might be able to do some vector transformations.

```
/* File : vector.h */

typedef struct {
double x;
double y;
double z;
double w;
} Vector;

/* File : vector.c */

#include "vector.h"

/* Make a new vector */

Vector *createv(double x, double y, double z, double w) {
  Vector *v;
  v = (Vector *) malloc(sizeof(Vector));
  v->x = x;
  v->y = y;
  v->z = z;
  v->w = w;
  return v;
}

/* Destroy vector */

void destroyv(Vector *v) {
  free(v);
}

/* Print a vector */
void printv(Vector *v) {
  printf("x = %g, y = %g, z = %g, w = %g\n",
 v->x, v->y, v->z, v->w);
}
```

```
/* Do a transformation */
void transform(double **m, Vector *v, Vector *r) {
  r->x = m[0][0]*v->x + m[0][1]*v->y + m[0][2]*v->z + m[0][3]*v->w;
  r->y = m[1][0]*v->x + m[1][1]*v->y + m[1][2]*v->z + m[1][3]*v->w;
  r->z = m[2][0]*v->x + m[2][1]*v->y + m[2][2]*v->z + m[2][3]*v->w;
  r->w = m[3][0]*v->x + m[3][1]*v->y + m[3][2]*v->z + m[3][3]*v->w;
}
```

Now we extend the `matrix.i` file to include our new functions :

```
/* FILE : matrix.i */

%init matrix_init
%{
#include <math.h>
#include "vector.h"
%}
extern double **new_matrix();
extern void destroy_matrix(double **);
extern void print_matrix(double **);
extern void set_m(double **, int, int, double);
extern double get_m(double **, int, int);
extern void mat_mult(double **a, double **b, double **c);
extern Vector *createv(double,double,double,double);
extern void destroyv(Vector *);
extern void printv(Vector *);
extern void transform(double **, Vector *, Vector *);
```

Compiling and running ...

```
% wrap -tcl matrix.i
% gcc matrix.c vector.c matrix_wrap.c main.c -ltcl -lm -o matrix
% matrix
matrix > source matrix.tcl
matrix > set M [new_matrix]
100086a8
matrix > identity $M
matrix > rotx $M 45
matrix > roty $M 30
matrix > scale $M 0.5
matrix > set v [createv 1.0 2.5 -3.0 1.0]
10009ac8
matrix > set t [createv 0 0 0 0];
10009a78
matrix > transform $M $v $t
matrix > printv $t
x = -0.316986, y = 1.97922, z = -0.211455, w = 0.5
matrix >
```

We have now extended our matrix program with a complex data type Vector and used it in Tcl. More importantly, none of our C code (except main) contains any Tcl specific variables or functions. Thus, we could just as easily use this C code from other C modules or without using the Tcl interface at all (note : this makes it extremely easy to prototype and debug code that may not even be used under a Tcl/Perl/Guile interface).

Eliminating main()

The `main()` function still contains Tcl specific code, but we can eliminate this as well by relying on the library capabilities. Suppose that we wanted to add these functions to `wish` instead. To do this, we would change the interface file as follows :

```
/* FILE : matrix.i */

%init matrix_init
%{
#include <math.h>
#include "vector.h"
%}
%include wish            // Add these to wish program

extern double **new_matrix();
extern void destroy_matrix(double **);
extern void print_matrix(double **);
extern void set_m(double **, int, int, double);
extern double get_m(double **, int, int);
extern void mat_mult(double **a, double **b, double **c);
extern Vector *createv(double,double,double,double);
extern void destroyv(Vector *);
extern void printv(Vector *);
extern void transform(double **, Vector *, Vector *);
```

The compilation process would now look like :

```
% wrap -tcl matrix.i
% gcc matrix.c vector.c matrix_wrap.c -ltk -ltcl -lX11 -lm  -o matrix
%
```

Afterwards, the program `matrix` is a fully functional `wish` application with our new matrix and vector operations added to it.

Documentation

Finally, let's add some documentation to our interface so we can remember what we actually did at a later date. This is done by adding C comments and few directives.

```
/*  FILE  :  matrix.i  */

%title "Matrix Example"
%init matrix_init
%{
#include <math.h>
#include "vector.h"
%}
%include wish            // Add these to wish program

%section "Matrix operations"
extern double **new_matrix();
/* Creates a new matrix and returns a pointer to it. */

extern void destroy_matrix(double **M);
/* Destroys the matrix M */

extern void print_matrix(double **M);
/* Prints out the matrix M */

extern void set_m(double **M, int i, int j, double val);
/* Sets M[i][j] = val */

extern double get_m(double **M, int i, int j);
/* Returns M[i][j] */

extern void mat_mult(double **a, double **b, double **c);
/* Multiplies matrix a by b and places the result in c */

%section "Vector operations"
extern Vector *createv(double x,double y,double z,double w);
/* Creates a new vector (x,y,z,w) */

extern void destroyv(Vector *v);
/* Destroys the vector v */

extern void printv(Vector *v);
/* Prints out vector v */

extern void transform(double **, Vector *, Vector *);
/* Transforms vector v to vector t, by M*v --> t */
```

Using the `-dascii` option produces the following documentation in the file `matrix_wrap.doc`.

```
Matrix Example
Created : Thu Feb 8 01:43:59 1996


1. Matrix Operations
====================

void : destroy_matrix M
          Destroys the matrix M

double : get_m M i j
          Returns M[i][j]

void : mat_mult a b c
          Multiplies matrix a by b and places the result in c

double ** : new_matrix
          Creates a new matrix and returns a pointer to it

void : print_matrix M
          Prints out the matrix M

void : set_m M i j val
          Sets matrix M[i][j] = val


2. Vector Operations
====================

Vector * : createv x y z w
          Creates a new vector (x,y,z,w)

void : destroyv v
          Destroys the vector v

void : printv v
          Prints out the vector v

void : transform T v t
          Transforms vector v to vector t, by M*v --> t
```

We can now look at this documentation file to see how our functions should be called when writing scripts at a later point.

13. Where to go from here?

That's about it. I have tried to present SWIG with a few simple examples, but the approaches described here can be extended to much more complicated situations. For more examples, I encourage you to look at the Examples directory in the SWIG source directory. You can also check out the SWIG homepage.

This document has only touched the surface of what one can do with SWIG. If you find this package useful, I'd like to hear your comments and suggestions on how to make it even better. I can be reached at beazley@cs.utah.edu.

Appendix A : Language Implementation Notes

A.1. Tcl/TK

SWIG has been developed for use with Tcl 7.4 and Tk 4.0. It can work with older versions too, but may need to slightly modify the SWIG library files “wish.c” and “tclsh.c” since the way in which these applications initialize user extensions has changed slightly.

Variable Linkage:

Variables with the following datatypes may be linked with Tcl.

- `int, unsigned int`
- `double`
- `char *`

Attempts to link with variables of other types are ignored.

Functions:

Functions may take arguments of the following types :

```
int, short, long, char, double, float, void, user_defined
```

Pointers may be used with all of the above types and the `signed` or `unsigned` modifier can be applied to integer types (`int, short, long, char`).

Functions may return the value of all of the above types except for user_defined types (structs and objects). Pointers to all of the above datatypes may also be returned.

Pointers :

Pointers are represented as hexadecimal character strings. SWIG uses it’s own functions for generating and reading these strings, but they should be compatible with the output of

```
printf(“%x”, (unsigned int) ptr);
```

A.2. Perl4

A general comment : Perl4 may somewhat easier to use than Perl5 if you don't care about Perl5's object oriented features. However, Perl5 seems to have a better mechanism for using user extensions.

Compiling:

To build a new Perl4 application, you will need to link with the file "uperl.o" which is found in the Perl4 source directory. You will also need to use the Perl source directory to include a few header files. Personally, I set up a symbolic link "`/usr/local/include/perl`" which points to the Perl source directory on my machine. This puts all of the needed files in a logical location.

Variable Linkage:

All variable types except user-defined types may be used. Pointers may also be used.

Functions:

Functions may take arguments of the following types :

```
int, short, long, char, double, float, void, user_defined
```

Pointers may be used with all of the above types and the `signed` or `unsigned` modifier can be applied to integer types (`int, short, long, char`).

Functions may return the value of all of the above types except for user_defined types (structs and objects). Pointers to all of the above datatypes may also be returned.

Pointers :

Pointers are represented as hexadecimal character strings usings the same mechanism as used with Tcl.

A.3. Perl5

I'm pleased to announce that SWIG works with Perl5 and can be used to produce dynamically loadable Perl5 modules. As of this writing, this is quite experimental, but early results are promising.

Normally, Perl 5 may be extended by writing function declarations in the XS language and using the `xsubpp` preprocessor supplied in the Perl5 distribution. SWIG completely replaces this system. However, extending Perl5 is somewhat more complicated that other languages so a few things need to be discussed :

Modules and Packages

Perl5 extensions can be dynamically loaded as modules. These modules may, in turn, be part of a larger package. The module name should be specified using the SWIG `%init` directive or on the command line using the `-module` option. For example, consider the earlier example :

```
/* File : example.i */
%init Example       // Module name
%{
/* Put headers and other declarations here */
%}

extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
```

Compiling

The interface file should now be processed with swig as follows :

```
%wrap -perl5 example.i
```

SWIG will produce 2 files. The first file, `example_wrap.c` contains the C code needed to link with Perl5 (this is roughly equivalent to the code that would have been produced by `xsubpp`). The second file is `Example.pm` which is needed to dynamically load the Example module. This file looks like the following :

```
package Example;
require Exporter;
require DynaLoader;
@ISA = qw(Exporter DynaLoader);
@EXPORT = qw( );
bootstrap Example;
Example::var_Example_init();
1;
```

By default, SWIG sets the package name to the same as the module name and does not export any symbols. Also, variables must be linked in after the bootstrap process. This is done by the call to Example::var_Example_init(). Variables are linked using the “Magic” mechanism described in the Perl5 documentation.

Rather than producing an executable, you should now produce a shared object file. This can be done as follows :

```
% gcc example.c -c
% gcc example_wrap.c -I/usr/local/src/perl5.001m -c
% ld -shared example.o example_wrap.o -o Example.so
```

Again, the name of the resulting shared object file is important. You’ll also have to include some headers in the Perl5 source directory.

Now runing our example :

```
% perl5
use Example;
print Example::fact(4),"\n"
24
%
```

Other Compilation Options :

The package name can be specified using the -package command line option as in :

```
% wrap -perl5 -package MyPackage example.i
```

All functions and variables will then be accessed as `MyPackage::fact()` for example.

You can export all of the functions into Perl's global name space by using

```
% wrap -perl5 -exportall example.i
```

This will allow you access functions without the package prefix. However, this practice is generally discouraged so don't say I didn't warn you.

The `-exportall` option only applies to functions. Variables will still have to be accessed using the package prefix. However, you might be able to import variables into the global namespace by calling the `Package::var_Module_init()` function manually in one of your scripts.

Variable Linkage:

Variable linkage follows the same rules as in the Perl4 implementation. Currently, pointers are still represented as strings and are completely incompatible with Perl5 references. In short, don't use a SWIG pointer as a Perl reference.

Functions :

Functions may take arguments involving all of the basic datatypes. However, don't pass Perl5 pointers because this will probably crash something.

General Comments :

The Perl5 port is still highly experimental. It needs alot of testing, and may be somewhat buggy. I'm still getting accustomed to the dynamic loader so I may have to rethink a few things. Suggestions are, as always, welcome.

A.4. Guile-iii

Guile is the GNU extension language based on Scheme. As of this writing, the system still seems to be undergoing development. If you're brave and like to play with experimental systems, I encourage you to pick up a copy.

Compiling:

You'll need to link with the library `libguile.a` and use several include files in the `guile` installation. I simply put symbolic links to these items in `/usr/local/include` and `/usr/local/lib` on my machine to make life easier. You'll also need to supply some sort of main program or interpreter. I have already done this and you can use it by putting an "`%include interpreter`" directive in your interface file.

Variable Linkage:

All variable types except user-defined types may be used. Pointers may also be used. I couldn't figure out a clean way to link variables in Guile (there doesn't seem to be any facilty for this that I could find). Therefore all variables are currently linked in as Guile functions taking an optional argument. These functions are used as follows :

```
(a)             ; Evalulates the variable a
(a 3.5)         ; Sets the variable a to 3.5
```

Functions:

Functions may take arguments of the following types :

```
int, short, long, char, double, float, void, user_defined
```

Pointers may be used with all of the above types and the `signed` or `unsigned` modifier can be applied to integer types (`int, short, long, char`).

Functions may return the value of all of the above types except for user_defined types (structs and objects). Pointers to all of the above datatypes may also be returned.

Pointers :

Pointers are represented as a character strings exactly as in the Tcl/Perl implementations.

A.5. The Future

There are many other possibilities. I'm working on a port for Matlab4.2 to support some scientific applications I'm working on. One can imagine supporting many other packages including iTcl, TkPerl, etc.... unfortunately, there is only a finite amount of time available....sigh.