Introduction 1

What is SWIG?

SWIG is a code development tool that makes it possible to quickly build powerful scripting language interfaces to C, C++, or Objective-C programs[1]. In a nutshell, SWIG is a compiler that takes C declarations and turns them into the "glue" needed to access them from common scripting languages including Perl, Python, and Tcl. SWIG usually requires no modifications to existing C code and can often be used to build a working interface in a matter of minutes. This makes it possible to do a number of interesting things including :

- Building powerful interfaces to existing C programs.
- Rapid prototyping and application development.
- Interactive debugging.
- Making a graphical user interface (using Tk for example).
- Powerful testing of C libraries and programs (using scripts).
- Building high performance C modules for scripting languages.
- Making C programming more enjoyable (or tolerable depending on your point of view)
- Impressing your friends.

There are some computer scientists who seem to believe that the only way to solve complicated problems is to create software of epic proportions and to have some sort of "grand" software design. Unfortunately this seems to lead to solutions that are even more complicated than the original problem. This, in turn enables the user to forget about the original problem and spend their time cursing at their machine (hence, the term "enabling technology"). SWIG, on the other hand, was developed because I was fed up with how much time I was wasting trying to develop flexible scientific applications. I wanted a tool that would let me use scripting languages to glue different things together, but didn't get in the way of the real problems I was working on. I wanted a simple tool that scientists and engineers could use to put together applications involving number crunching, data analysis, and visualization without having to worry about tedious systems programming, making substantial modifications to existing code, trying to figure out a big monolithic computing "environment," or having to get a second degree in computer science. In short, I wanted to have a tool that would improve application development, but stay out of the way as much as possible.

Life before SWIG

SWIG was developed to make my life easier as a C programmer. While C is great for high perfor-

1. For the purposes of documentation, a "C-program" means either ANSI C, C++, or Objective-C.

mance and systems programming, trying to make an interactive and highly flexible C program is a nightmare (in fact, it's much worse than that). The real problem is that for every C program I wrote, I needed to have some sort of interface, but being more interested in other problems, I would always end up writing a really bad interface that was hard to extend, hard to modify, and hard to use. I suppose I could have tried to do something fancy using X11, but who has time to waste weeks or months trying to come up with an interface that is probably going to end up being larger than the original C code? There are more interesting problems to work on.

The real problem, perhaps, is that most C programs end up being structured as follows :

- A collection of functions and variables.
- A `main()` program that starts everything up.
- A bunch of hacks added to make it usable.

The `main()` program may be written to handle command line arguments or to read data from `stdin`, but either way, modifying or extending the program to do something new requires changing the C code, recompiling, and testing. If you make a mistake, you need to repeat this cycle until things work. Of course, as more and more features are added, your C program turns into a hideous unintelligible mess that is even more difficult to modify than it was before (Of course, if you're lucky, your project starts out as an unintelligible mess).

Life after SWIG

With SWIG, I was hoping to avoid many of the headaches of working with C programs, by structuring things as follows :

- A collection of functions and variables.
- A nice interpreted interface language that can be used to access everything.

With this model, you keep all of the functionality of your C program, but access it through a scripting language interface instead of writing more C code. This is nice because you are given full freedom to call functions in any order, access variables, and write scripts to do all sorts of interesting things. If you want to change something, just modify a script, and rerun it. If you're trying to debug a collection of functions, you can call them individually and see what they do. If you're trying to build a package out of various components, you can just glue everything together with a scripting language and have a common interface to all of the various pieces.

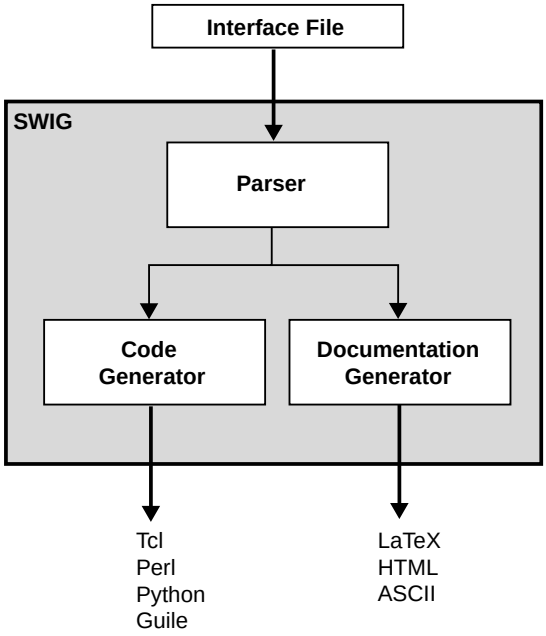
SWIG tries to make the integration between scripting languages and C as painless as possible. This allows you to focus on the underlying C program and using the high-level scripting language interface, but not the tedious and complex chore of making the two languages talk to each other.

I like this model of programming. While it's certainly no "silver-bullet", it has made programming alot more enjoyable and has enabled us to solve complicated problems that would have been unneccessarily difficult using only C.

The SWIG package

SWIG is a compiler that takes ANSI C declarations and turns them into a file containing the C

code for binding C functions, variables, and constants to a scripting language (SWIG also supports C++ class and Objective-C interface definitions). Input is specified in the form of an "interface file" containing declarations (input can also be given from C source files provided they are sufficiently clean). The SWIG parser takes this input file and passes it on to a code generation and documentation module. These modules produce an interface for a particular scripting language along with a document describing the interface that was created. Different scripting languages are supported by writing new back-end modules to the system.



A SWIG example

The best way to illustrate SWIG is with a simple example. Consider the following C code:

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

double  My_variable  = 3.0;

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

/* Compute n mod m */
int my_mod(int n, int m) {
       return(n % m);
}
```

Suppose that you wanted to access these functions and the global variable `My_variable` from Tcl. We start by making a SWIG interface file as shown below (by convention, these files carry a .i suffix) :

SWIG interface file

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
%}

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

The interface file contains ANSI C function prototypes and variable declarations. The `%module` directive defines the name of the module that will be created by SWIG. The `%{,%}` block provides a location for inserting additional code such as C header files or additional C functions.

The swig command

SWIG is invoked using the `swig` command. We can use this to build a Tcl module (under Linux) as follows :

```
unix > swig -tcl example.i
Generating wrappers for Tcl.
unix > gcc -c -fpic example.c example_wrap.c -I/usr/local/include
unix > gcc -shared example.o example_wrap.o -o example.so
unix > tclsh
% load ./example.so
% fact 4
24
% my_mod 23 7
2
% expr $My_variable + 4.5
7.5
%
```

The `swig` command produced a new file called `example_wrap.c` that should be compiled along with the `example.c` file. Most operating systems and scripting languages now support dynamic loading of modules. In our example, our Tcl module has been compiled into a shared library that can be loaded into Tcl and used. When loaded, Tcl will now have our new functions and variables added to it. Taking a careful look at the file `example_wrap.c` reveals a hideous mess, but fortunately you almost never need to worry about it.

Building a Perl5 module

Now, let's turn these functions into a Perl5 module. Without making any changes type the following (shown for Solaris):

```
unix > swig -perl5 example.i
Generating wrappers for Perl5
unix > gcc -c example.c example_wrap.c \
       -I/usr/local/lib/perl5/sun4-solaris/5.003/CORE
unix> ld -G example.o example_wrap.o -o example.so       # This is for Solaris
unix > perl5.003
use example;
```

```
print example::fact(4), "\n";
print example::my_mod(23,7), "\n";
print $example::My_variable + 4.5, "\n";
<ctrl-d>
24
2
7.5
unix >
```

Building a Python module

Finally, let's build a module for Python1.4 (shown for Irix).

```
unix > swig -python example.i
Generating wrappers for Python
unix > gcc -c example.c example_wrap.c -I/usr/local/include/python1.4
unix > ld -shared example.o example_wrap.o -o examplemodule.so     # Irix
unix > Python
Python 1.4 (Sep 10 1996)  [GCC 2.7.0]
Copyright 1991-1996 Stichting Mathematisch Centrum,
Amsterdam
>>> import example
>>> example.fact(4)
24
>>> example.my_mod(23,7)
2
>>> example.cvar.My_variable + 4.5
7.5
```

Shortcuts

To the truly lazy programmer, one may wonder why we needed the extra interface file at all. As it turns out, we can often do without it. For example, we could also build a Perl5 module by just running SWIG on the C source and specifying a module name as follows

```
% swig -perl5 -module example example.c
unix > gcc -c example.c example_wrap.c \
       -I/usr/local/lib/perl5/sun4-solaris/5.003/CORE
unix> ld -G example.o example_wrap.o -o example.so
unix > perl5.003
use example;
print example::fact(4), "\n";
print example::my_mod(23,7), "\n";
print $example::My_variable + 4.5, "\n";
<ctrl-d>
24
2
7.5
```

Of course, there are some restrictions as SWIG is not a full C/C++ parser. If you make heavy use of the C preprocessor, complicated declarations, or C++, giving SWIG a raw source file probably isn't going to work very well (in this case, you would probably want to use a separate interface file).

SWIG also supports a limited form of conditional compilation. If we wanted to make a combination SWIG/C header file, we might do the following :

```
/* File : example.h */
#ifdef SWIG
%module example
%include tclsh.i
#endif
extern double My_variable;
extern int    fact(int);
extern int    my_mod(int n, int m);
```

Documentation generation

In addition to producing an interface, SWIG also produces documentation. For our simple example, the documentation file may look like this :

```
example_wrap.c

[ Module : example, Package : example ]

$My_variable
        [ Global : double My_variable ]

fact(n);
        [ returns int  ]

my_mod(n,m);
        [ returns int  ]

get_time();
        [ returns char * ]
```

C comments can be used to provide additional descriptions. SWIG can even grab these out of C source files in a variety of ways. For example, if we process `example.c` as follows :

```
swig -perl5 -Sbefore -module example example.c
```

We will get a documentation file that looks like this (with our C comments added) :

```
example_wrap.c

[ Module : example, Package : example ]


$My_variable
        [ Global : double My_variable ]

fact(n);
        [ returns int  ]
        Compute factorial of n

my_mod(n,m);
        [ returns int  ]
        Compute n mod m
```

Building libraries and modules

In addition to generating wrapper code, SWIG provides extensive support for handling multiple files and building interface libraries. For example, our `example.i` file, could be used in another interface as follows :

```
%module foo
%include example.i            // Get definitions from example.i

... Now more declarations ...
```

In a large system, an interface might be built from a variety of pieces like this :

```
%module package

%include network.i
%include file.i
%include graphics.i
%include objects.i
%include simulation.i
```

SWIG comes with a library of existing functions known as the SWIG library. The library contains a mix of language independent and language dependent functionality. For example, the file '`array.i`' provides access to C arrays while the file '`wish.i`' includes specialized code for rebuilding the Tcl wish interpreter. Using the library, you can use existing modules to build up your own personalized environment for building interfaces. If changes are made to any of the components, they will appear automatically the next time SWIG is run.

C syntax, but not a C compiler

SWIG uses ANSI C syntax, but is not a full ANSI C compiler. By using C syntax, I hope to make SWIG easy to use with most C programs, easy to learn, and easy to remember. Other tools tend to use a precise interface definition language, but I personally find this approach to be painful. When I want to build an interface to a collection of several hundred C functions, I don't necessarily want to write a special interface definition for each one. Nor do I want to have to go dig up the manual because I can't remember the syntax.

On the other hand, using C syntax can be ambiguous. For example, if you have the following declaration

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

We don't really know if `a` is an array of some fixed size, a dynamically allocated block of memory, a single value, or an output value of the function. For the most part, SWIG takes a literal approach (`a` is obviously a `double *`) and leaves it up to the user to use the function in a correct manner. Fortunately, there are several ways to help SWIG out using additional directives and "helper" functions. Thus, the input to SWIG is often a mix of C declarations, special directives and hints. Like Perl programming, there is almost always more than one way to do almost anything.

SWIG does not currently parse every conceivable type of C declaration that it might encounter in a C/C++ file. Many things may be difficult or impossible to integrate with a scripting language

(C++ templates for example). Thus, SWIG may not recognize advanced C/C++ constructs. I make no apologies for this--SWIG was designed to access C, but was never intended to magically turn scripting languages into some sort of bizarre C++ interpreter. Of course, SWIG's parser is always being improved so currently unsupported features may be supported in later releases.

Non-intrusive interface building

When used as I intended, SWIG requires minimal modification to existing C code. This makes SWIG extremely easy to use with existing packages, but also promotes software reuse and modularity. By making the C code independent of the high level interface, you can change the interface and reuse the code in other applications. In a similar spirit, I don't believe that there is any one "best" scripting language--use whichever one you like (they're all pretty good). There's no real reason why a particular application couldn't support multiple scripting language interfaces to serve different needs (or to have no scripting language interface at all).

Hands off code generation

SWIG is designed to produce working code that needs no hand-modification (in fact, if you look at the output, you probably won't want to modify it). Ideally, SWIG should be invoked automatically inside a Makefile just as one would call the C compiler. You should think of your scripting language interface being defined entirely by the input to SWIG, not the resulting output file. While this approach may limit flexibility for hard-core hackers, it allows others to forget about the low-level implementation details. This is my goal.

Event driven C programming

By adding a scripting language interface to a program, SWIG encourages an event-driven style of programming (although it may not be immediately obvious). An event driven program basically consists of a large-collection of functions (called callback functions), and an infinite loop that just sits and waits for the user to do something. When the user does something like type a command, hit a key, or move the mouse the program will call a function to perform an operation--this is an event. Of course, unless you've been living in cave for the last 20 years, you've used this kind of approach when running any sort of graphical user interface.

While you may not be using SWIG to develop a GUI, the underlying concept is the same. The scripting language acts as an event-loop waiting for you to issue commands. Unlike a traditional C application (that may use command line options), there is no longer a fixed sequence of operations. Functions may be issued in any order and at any time. Of course, this flexibility is exactly what we want!

However, there are a number of things to keep in mind when developing an application with SWIG :

- Functions may be called at any time and in any order. It is always a good idea for functions to check their arguments and internal state to see if it's legal to proceed (Not doing so usually results in mysterious crashes later on).
- Code should be structured as a collection of independent modules, not a huge mess of

interrelated functions and variables (ie. spaghetti code).
- Global variables should be used with care.
- Careful attention to the naming of variables and functions may be required to avoid namespace conflicts when combining packages.

While it may be hard (or impossible) to address all these problems in a legacy code, I believe that using SWIG encourages all of the above qualities when developing new applications. This results in code that is more reliable, more modular, and easier to integrate into larger packages. By providing a non-intrusive, easy to use tool, it is possible to develop highly reliable event-driven code from the start--not as a hack to be added later. As a final sales pitch, in the initial application for which SWIG was developed, code reliability and flexibility has increased substantially while code size has decreased by more than 25%. I believe this is a good thing.

Automatic documentation generation

SWIG makes it very easy to build large interactive C programs, but it can sometimes be hard to remember what C functions are available and how they are called in the scripting language interface. To address this problem, SWIG automatically generates a documentation file in a number of different formats. C comments can be used to provide additional descriptions of each function, and documentation can be grouped into a hierarchy of sections and subsections. The documentation file is intended to provide a reasonable description of the scripting language interface. While it's no competition for a full-blown C code documentation generator, the documentation system can do a reasonable job of documenting an interface.

Summary

At this point, you know about 95% of everything you need to know to start using SWIG. First, functions and variables are specified using ANSI C, C++, or Objective-C syntax. These may appear in a separate interface file or you can use a C source file (if it is sufficiently clean). SWIG requires no modifications to existing C code so it's easy to get started. To build a module, use the `swig` command with an appropriate target language option. This generates a C file that you need to compile with the rest of your code and you're ready to go.

I don't consider there to be a "right" or "wrong" way to use SWIG, although I personally use separate interface files for a variety of reasons :

- It helps keep me organized.
- It's usually not necessary to wrap every single C function in a program.
- SWIG provides a number of directives that I tend to use alot.
- Having a special interface file makes it clear where the scripting language interface is defined. If you decide to change something in the interface, it's easy to track down if you have special file.

Of course, your mileage may vary.

SWIG for Windows and Macintosh

SWIG has been primarily developed and designed to work with Unix-based applications. How-

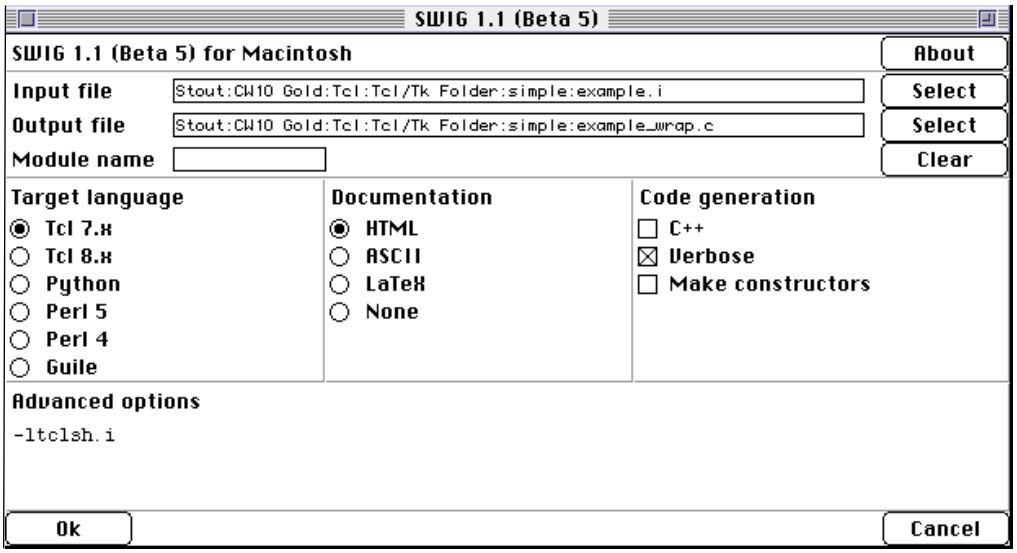
ever, most of the scripting languages supported by SWIG are available on a variety of other platforms including Windows 95/NT and Macintosh. SWIG generated code is mostly compatible with these versions (and SWIG itself can now run on these platforms).

SWIG on Windows 95/NT

The Windows 95/NT port of SWIG (provided by Kevin Butler), is a straightforward translation of the Unix version. At this time it is only known to work with Microsoft Visual C++ 4.x, but there is nothing to prevent its use with other compilers (at least not that I'm aware of). SWIG should be invoked from the MS-DOS prompt in exactly the same manner as in Unix. SWIG can also be used in NMAKE build files and added to Visual C++ projects under the custom build setting. As of this writing, SWIG is known to work with Windows versions of Tcl/Tk, Python, and Perl (including the ActiveWare Perl for Win32 port). SWIG makes extensive use of long filenames, so it is unlikely that the SWIG compiler will operate correctly under Windows 3.1 or DOS (the wrapper code generated by SWIG may compile however).

SWIG on the Power Macintosh

A Macintosh port of SWIG is also available, but is highly experimental at this time. It only works on PowerPC based Macintosh systems running System 7 or later. Modules can be compiled with the Metrowerks Code Warrior compiler, but support for other compilers is unknown. Due to the lack of command line options on the Mac, SWIG has been packaged in a Tcl/Tk interface that allows various settings and command line options to be specified with a graphical user interface. Underneath the hood, SWIG is identical to the Unix/Windows version. It recognizes the same set of options and generates identical code. Any SWIG command line option listed in this manual can be given to the Mac version under "Advanced Options" shown in the figure. At this writing, SWIG is only known to support Mac versions of Tcl/Tk. Work on Macintosh versions of Python and Perl is underway.



Cross platform woes

While SWIG and various freely available scripting languages are supported on different plat-

forms, developing cross platform applications with these tools is still immature and filled with pitfalls. In fact, it's probably only recommended for true masochists. Despite this, it's interesting to think about using freely available tools to provide common interfaces to C/C++ code. I believe that SWIG may help, but it is by no means a cross-platform solution by itself.

How to survive this manual

This manual was written to support the Unix version of SWIG. However, all of the main concepts and usage of SWIG also apply to the Windows and Macintosh versions. You should be forewarned that most of the examples are Unix-centric and may not compile correctly on other machines. However, most of the examples provided with the SWIG distribution have now been tested under both Unix and Windows-NT. When applicable, I will try to point out incompatibilities, but make no promises...