
Using Perl with Pro/Toolkit

Marc Mettes
BIC Corp, Milford, CT
Marc.Mettes@BicWorld.com
203-783-2721

Using Perl with Pro/Toolkit

- **What do you mean?**
- **Why bother?**
- **How is it done?**
- **Conclusions**

What do you mean?

- **Calling Pro/Toolkit Libraries from Perl**
 - This is called 'Extending Perl'
 - Hard to do, time consuming, but easy to use in Perl
- **Utilizing the Perl interpreter from Pro/Toolkit**
 - This is called 'Embedding Perl'
 - Easy to do, but hard to use
 - Not absolutely necessary for spawn mode apps or asynchronous, but hard to avoid
 - Absolutely necessary for dll mode apps
- **Documented on the Internet**
 - Many pieces of documentation exist for embedding and extending Perl.
 - Unfortunately, you have to read them all!

Why use Perl with Pro/Toolkit?

- **Same benefits with using Perl**
 - **Rapid software development**
 - **Excellent multi–platform support**
 - **Unix, Linux, Windows, Mac, Amiga, etc.**
 - **Regular expression pattern matching**
 - **Object oriented (or not)**
 - **Huge ready–to–use library available on CPAN**
 - **Text and Data Processing**
 - **Networking Protocols (HTTP, FTP, TCP, etc.)**
 - **Database Connectivity (Oracle, DB2, etc.)**
 - **Alternative GUIs**
 - **Automatic garbage collection**
 - **Dynamic Language**
 - **Can rewrite itself on the fly.**
 - **Free**

How is it done?

- **Embedding**
 - **See `perlembed` man page for starters.**
 - **`http://perldoc.com`**
 - **Avoidable for spawn and asynchronous apps, but complicated**
 - **`protk.dat` file requires executable**
 - **Will not accept script or batch file**
 - **Standard Perl invocation requires command line args**
 - **Command line args cannot be used in `protk.dat`**
 - **Not avoidable for dll mode apps**
 - **The application must have access to a perl interpreter**
 - **Solutions**
 - **Embed**
 - **Rewrite the perl interpreter main program**
 - **Execute a known perl script**
 - **Wrap**
 - **Create wrapper executable**
 - **Run perl script from wrapper via `system()` or `exec()`**

How is it done?

- Perl interpreter main program:

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *my_perl;
EXTERN_C void xs_init();

int main(int argc, char **argv, char **env) {
    char *perl_argv[] = { "interp", "interp.pl" };
    int perl_argc = 2;
    char *args[] = { NULL };
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, xs_init, perl_argc, perl_argv,
               (char **)NULL);
    ProToolkitMain(argc, argv); /* Calls user_initialize() */
}
```

- **Perl interpreter main program:**

- **`user_initialize()` (*calls perl `user_initialize()` from C*)**
- **Must be in the interpreter main program**

```
int user_initialize() {  
    char *args[] = { NULL };  
    call_argv("user_initialize", G_DISCARD | G_NOARGS, args);  
    return(0);  
}
```

How is it done?

- **Extending Perl**

- **Use the provided extension tools**

```
% h2xs -cn ProETK
```

- **h2xs will create the starting point for you including**

- **ProETK.xs Extension C Code**
- **ProETK.pm Perl Module File**
- **Makefile.PL Makefile Generator**

- **Change Makefile.PL**

- **Change LIBS to something like this (ignore word wrap):**

```
'LIBS' => ['-lsocket -lnsl -lw -lm -ldl  
          -Lprotoolkit/sun4_solaris/obj -lprotoolkit']
```

- **Change INCS to something like this (ignore word wrap):**

```
'INCS' => ['-protoolkit/includes']
```

- **Hello World Example**

- **ProETK.xs:**

- **ProMessageDisplay()** (*Available from perl*)

```
MODULE = ProETK
```

```
PACKAGE = ProETK
```

```
void
```

```
ProMessageDisplay(msg_file, format_string, mesg_string)
```

```
char *msg_file
```

```
char *format_string
```

```
char *mesg_string
```

```
PREINIT:
```

```
ProFileName w_msg_file;
```

```
CODE:
```

```
ProStringToWstring(w_msg_file, msg_file);
```

```
ProMessageDisplay(w_msg_file, format_string, mesg_string);
```

- Hello World Example

- Perl script contents:

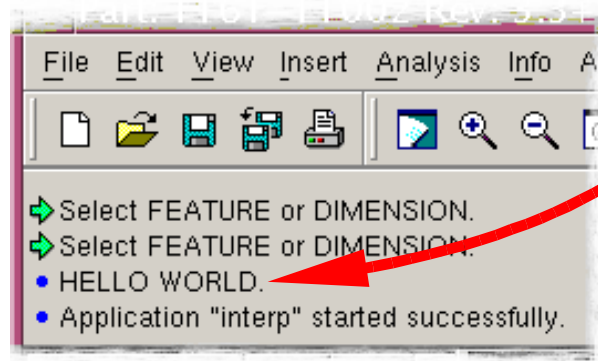
```
use ProETK;
```

```
sub user_initialize {
```

```
    ProETK::ProMessageDisplay(
```

```
        "msg_file.txt", "USER %0s", "HELLO WORLD.");
```

```
}
```



- ProMdl Example (void pointers)
 - We can create an object oriented interface
 - \$mdl1 = new ProMdl;

```
MODULE = ProETK          PACKAGE = ProMdl
```

```
ProMdl
new(class, ...)
  char * class
PREINIT:
  ProMdl model;
  ProCharName name;
  ProName w_name;
  ProMdlType type;
CODE:
  if (items == 3) {
    strcpy(name, SvPV(ST(1), PL_na));
    type = SvIV(ST(2));
    ProStringToWstring(w_name, name);
    ProMdlInit(w_name, type, &model);
  }
  else if (items == 1) {
    ProMdlCurrentGet(&model);
  }
  RETVAL = model;
OUTPUT:
  RETVAL
```

- **ProMdl Example (void pointers) ... continued**
 - **Class method are defined within the package**

```
char *
ProMdlNameGet(model)
    ProMdl model
PREINIT:
    ProCharName tmpbuf;
    ProName w_tmpbuf;
    wchar_t *ptr;
CODE:
    ProMdlNameGet(model, w_tmpbuf);
    ProWstringToString(tmpbuf, w_tmpbuf);
    RETVAL = tmpbuf;
OUTPUT:
    RETVAL
```

- **ProMdl datatype requires a perl typemap entry**
 - Typemap tells perl how to represent data
 - How to handle data coming from C to Perl
 - How to handle data coming from Perl to C

```
ProMdl          T_PTROBJ
```

- **ProMdl will be treated as a Perl object**

- ProMdl Example (void pointers) ... continued

- This Perl code

```
$MdlRef = new ProMdl;  
print "MdlRef: $MdlRef", "\n";  
print "ProMdlNameGet: ", $MdlRef->ProMdlNameGet(), "\n";
```

ProMdl Object
Creation



- Produces this output

```
MdlRef: ProMdl=SCALAR(0x420a50)  
ProMdlNameGet: 1161-11002
```

Method Invocation



Indicates a Perl
ProMdl Object



- **ProModelitem Example (C struct)**
 - **We have to deal with getting the C struct into Perl**
 - **Copy struct into Perl scalar**
 - **Not appropriate for structs containing pointers**
 - **Probably not very efficient**
 - **malloc() and pass pointer to Perl object**
 - **Typemap entry required (needs tab characters):**

`ProModelitem * T_PTROBJ`

- **(ProModelitem *) will be treated as a Perl object**

- ProModelitem Example (C struct)

- ProETK.xs:

```
MODULE = ProETK      PACKAGE = ProModelitem
```

```
ProModelitem *  
new(class,...)  
  char * class
```

```
PREINIT:
```

```
  ProMdl model;  
  ProModelitem *model_item;  
  ProType prototype;  
  int item_id;
```

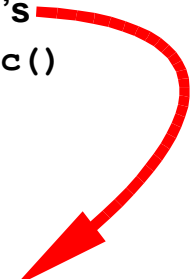
```
CODE:
```

```
  ProMdlCurrentGet(&model);  
  New(0, model_item, 1, ProModelitem);  
  ProMdlToModelitem(model,model_item);  
  RETVAL = model_item;
```

```
OUTPUT:
```

```
  RETVAL
```

New() is Perl's
portable malloc()



- ProModelitem Example (C struct)
 - If we allocate memory, we have to free it
 - Object oriented functionality helps here
 - Class method DESTROY () is called automatically by Perl

```
MODULE = ProETK      PACKAGE = ProModelitemPtr
```

```
void  
DESTROY (model_item)  
    ProParameter *model_item
```

```
CODE:  
    printf("Destroying ProModelitem\n");  
    Safefree(model_item);
```

Note different package name!

```
ProMdl  
ProModelitemMdlGet (model_item)  
    ProModelitem *model_item
```

```
PREINIT:  
    ProMdl model;  
CODE:  
    ProModelitemMdlGet (model_item, &model);  
    RETVAL = model;
```

Safefree () is Perl's portable free ()

```
OUTPUT:  
    RETVAL
```

- ProModelitem Example ... continued

- This Perl code

```
my $MI = new ProModelitem;  
print "MI: |$MI|", "\n";  
my $Mdl1 = $MI->ProModelitemMdlGet;  
print "Mdl1: |$Mdl1|", "\n";  
print "ProMdlNameGet: ", $Mdl1->ProMdlNameGet(), "\n";  
undef $MI;
```

ProModelitem
Object Creation

- Produces this output

```
MI: |ProModelitemPtr=SCALAR(0x42087c) |  
Mdl1: |ProMdl=SCALAR(0x42ed94) |  
ProMdlNameGet: 1161-11002  
Destroying ProModelitem
```

Force Perl garbage
collection

Indicates a Perl
ProModelitemPtr
Object

Garbage collection
has occurred

Conclusion

- **Perl can use Pro/Toolkit and vice versa**
 - This is just a proof of concept
- **To do?**
 - GUI stuff
 - Callbacks
 - Visit Functions
- **Want to help?**
 - Call me.