User Defined Extensions

From OpenGrads Wiki

Contents

Important

The instructions in this page are for GrADS v1.9.0-rc1. For the OpenGrADS Bundle v2.0.a5.oga.3 and later the OpenGrADS extensions are already included and require no installation. The same is true for any the *Win32 Superpacks*, including v1.9.0-rc1.

Additional information about the OpenGrADS User Defined Extensions for GrADS v2.0 will be posted here soon.

Introduction

One of the core activities in OpenGrADS is the development of high performance user defined commands and functions. The classic user defined functions (UDFs) in GrADS relies on disk files for the transmission of data to and from the UDF. While this approach is somewhat robust and language independent, reading the exchange file is somewhat tedious and the unnecessary I/O leads to a large

performance penalty.

Installing Pre-compiled User Defined Extensions

Requirements

GrADS v1.9.0 or later, installed

Download

These are available on our sourceforge download page (http://sourceforge.net /project/showfiles.php?group_id=161773&package_id=256757).

Installation

First download and install GrADS itself. The GrADS executables are typically placed in the directory `/usr/local/bin`. The libraries contained in this tar file are typically placed in the directory `/usr/local/bin/gex`. If you do not have write permission for your `/usr/local/bin` directory, you can put them in the `~/bin/gex` subdirectory of your home directory or any other directory of your choice. Henceforth, we will refer to the GrADS installation directory as `$GABIN`, and assume that you install the dynamic extensions in subdirectory `gex/`.

Whatever location you choose to install the UDXTs you must set the environment variable GAUDXT to point to the location of your User Defined Extension Table. To use the one in this distribution set

```
export GAUDXT=$GABIN/gex/udxt    (sh, ksh, bash)
setenv GAUDXT $GABIN/gex/udxt    (csh, tcsh)
```

You may want to familiarize yourself with the contents of the file "$GABIN/gex/udx" and comment out any function that may cause any conflict to you.

In addition, you need to let your operating system know where to find these shared objects (dynamic libraries). Typically

- **Linux, most Unices:**

```
export LD_LIBRARY_PATH=$GABIN/gex:$LD_LIBRARY_PATH    (sh, ksh, bash)
setenv LD_LIBRARY_PATH $GABIN/gex:$LD_LIBRARY_PATH    (csh, tcsh)
```

- **IRIX64 Note:**

You man want to set the environment variable LD_LIBRARYN32_PATH for New 32-bit ABI and LD_LIBRARY64_PATH for 64-bit programs, in addition the good old LD_LIBRARY_PATH. If this still does not work, enter the full pathnames in your "udxt" file.

- **Mac OS X**

```
export DYLD_LIBRARY_PATH=$GABIN/gex:$DYLD_LIBRARY_PATH  (sh, ksh, bash)
setenv DYLD_LIBRARY_PATH $GABIN/gex:$DYLD_LIBRARY_PATH  (csh, tcsh)
```

Verifying your installation

Start GrADS and enter

```
 ga-> query udct
 ga-> query udft
```

To see a list of all your user defined commands and functions. You may want to try

```
 ga-> hello
```

which should print "Hello, World" to your screen.

Building User Defined Extensions from Sources

Although in principal you do not need to build GrADS to build the extensions, it does gives flexibility to have the full sources around. The most convenient way is to check out module `Grads` (notice the capital "G"), which includes the standard GrADS sources as well as the extra `extensions` directory:

```
% cd workspace
% cvs -d ... co -P Grads  # Notice capital "G"
```

You can ensure maximum compatibility between your UDXTs and the GrADS binaries they extend by building GrADS at the same time; for instructions, see Building GrADS from Sources. In the very least you should configure your source tree

```
% cd Grads
% ./configure
```

For compiling and creating a tarball with your extensions enter

```
% make gex-dist
```

For simply building it,

```
% cd extensions
% make
% make install bindir=/path/to/bin/dir
```

You can also build from the individual directories,

```
% cd hello
% make
```

The GNUmakefiles for the individual packages include the fragment `gex.mk` and does not yet make uso of `automake`. Future enhancements include compilers wrappers such as `gexcc, gexf77` and `gexf90`.

For adding your own packages, create a directory under `extensions/` and copy over the GNUmakefile from one of the other packages to your newly created directory. The `hello` package is an excellent place to start.

Writing your own User Defined Extensions

This section has not been written yet. In the meantime, examine the available sources under `extensions/` as discussed in the previous section. Feel free to complete the documentation.

Background: How GrADS loads its extensions

User Defined Extensions in GrADS are loaded the same way web browsers load their plug-ins. It makes use of the `dlopen, dlsym()` and `dlclose()` function calls, a device introduced by Sun Microsystems (http://en.wikipedia.org /wiki/Sun_microsystems) back in the 1980's that are available in most modern platforms. Let's look at a simple example that loads the `cos` function from the standard C math library, `libm.so`:

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

```
int main(int argc, char **argv) {
 void *handle;
 double (*cosine)(double);
 char *error;
```

```
/* Open the existing libray libm.so */
handle = dlopen ("libm.so", RTLD_LAZY); /* use libm.dylib on a Mac */
if (!handle) {
  fprintf (stderr, "%s\n", dlerror());
  exit(1);
}
 /* Retrieve a pointer to the function "cos" provided by libm */
dlerror();    /* Clear any existing error */
*(void **) (&cosine) = dlsym(handle, "cos");
if ((error = dlerror()) != NULL) {
  fprintf (stderr, "%s\n", error);
  exit(1);
}
/* Print result and close the library */
printf ("%f\n", (*cosine)(0.2));
dlclose(handle);
return 0;
}
```

The function `dlopen()` opens a shared library, while `dlsym()` returns a function pointer given a string with the name of a function in the library; you can guess what `dlclose()` does. Do a `man dlopen` for information on the other parameters. The code fragment above opens the math library `libm.so`, get a pointer to the cosine function `cos` and executes it. If you save this code fragment in file `test.c`, you usually compile and run it like this

```
% gcc -o test test.c -ldl
% ./test
0.980067
```

Now, you do not have to rely on system libraries, you can write your own loadable libraries. Consider this simple implementation of the cosine function keeping the leading terms of its Taylor series (http://en.wikipedia.org/wiki/Taylor_series):

```
double cos ( double x ) {
   return (double) ( 1.0 - x*x*(0.5 - x*x/24.) );
}
```

If one saves this code fragment to a file called `mycos.c`, on many sytems, including Linux, a shared library (dynamically linked library) can be create with the command line

```
% gcc -shared -o mycos.so mycos.c
```

The particular command for creating a shared library varies from system to system. On Mac OS X, the syntax is

```
% gcc -c mycos.c
```

```
% libtool -dynamic -o mycos.dylib mycos.o
```

The OpenGrADS build mechanism can produce shared libraries for the most common systems. However, if you need to port extensions to a new unsupported system, getting this simple example to work is a good starting point.

Now that you have your own implementation of the cosine function, all you need to do is to change one line in the `test.c` sample code above (in a real application you would read these names from a file). Simply replace the line

```
handle = dlopen ("libm.so", RTLD_LAZY);
```

with this one

```
handle = dlopen ("./mycos.so", RTLD_LAZY);
```

Recompile and run:

```
% gcc -o test test.c -ldl
% ./test
0.980067
```

This simple example illustrates how a user defined function can be incorporated in an application at run time. The name of the library and function does not have to be known at compile time, it could be read from an external file. This is precisely how we implemented UDXTs in GrADS. A text file, the so-called UDXT table, contains the name of shared libraries and functions within. GrADS loads this table at start up, but loads an extension only upon the first usage. We ellaborate on these points next.

Writing extensions in C

Writing extensions in Fortran

Producing documentation for your extensions

Although not a strict requirement, we have been using 'Perl On-line Documentation' (http://perldoc.perl.org/perlpod.html) (POD) mark up syntax for documenting the contributed the user defined extensions directly at the source code. POD is a simple and yet adequate mark up language for creating basic Unix style man pages, and there are converters to html, MediaWiki, etc. In adittion, the `perldoc` utility can be used to display this documentation on the screen, e.g.,

```
% perldoc re
```

Or else, run this file through `cpp` to extract the POD fragments:

```
% cpp -DPOD -P < re.c > re.pod
```

and place `re.pod` in a place perldoc can find it, like somewhere in your path. To generate HTML documentation:

```
% pod2html --header < re.pod > re.html
```

To generate MediaWiki documentation:

```
% pod2wiki --style mediawiki < re.pod > re.wiki
```

If you have *pod2html* and *pod2wiki* installed on your system (if not, get them from CPAN (http://search.cpan.org), there are targets in the gex.mk fragment for these:

```
% make re.html
% make re.wiki
```

For having POD documentation generated (ans installed) automatically add this to your GNUmakefile:

```
export PODS=re.pod
```

(of course, replace `re` with the actual name of your file containing the POD documentation.) See the *re* UDXT for a example of POD documentation built right in the source code. The POD documentation is installed along with the GrADS binaries so that `perldoc` can access it automatically.

Documentation for Contributed User Defined Extensions

Consult the Manual Pages (http://opengrads.org/doc) for documentation about the individual User Defined Extensions.

- This page was last modified on 26 March 2009, at 19:13.